

Templates: Prototype document functions^{*}

Frank Mittelbach, Chris Rowley, David Carlisle, L^AT_EX Project[†]

October 19, 2026

1 Introduction

There are three broad “layers” between putting down ideas into a source file and ending up with a typeset document. These layers of document writing are

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with T_EX programming) of the design.

We write the text as an author, and we see the visual output of the design after the document is generated; the T_EX implementation in the middle is the glue between the two.

L^AT_EX’s greatest success has been to standardize a system of mark-up that balances the trade-off between ease of reading and ease of writing to suit almost all forms of technical writing. It’s other original strength was a good background in typographical design; while the standard L^AT_EX 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, L^AT_EX 2_ε has always lacked a standard approach to customising the visual design of a document. Changing the looks of the standard classes involved either:

- Creating a new version of the implementation code of the class and editing it.
- Loading one of the many packages to customise certain elements of the standard classes.
- Loading a completely different document class, such as KOMA-Script or memoir, that allows easy customization.

All three of these approaches have their drawbacks and learning curves.

The idea behind `lttemplates` is to cleanly separate the three layers introduced at the beginning of this section, so that document authors who are not programmers can easily change the design of their documents. `lttemplates` also makes it easier for L^AT_EX programmers to provide their own customizations on top of a pre-existing class.

^{*}This file has version v1.0n dated 2026-10-17, © L^AT_EX Project.

[†]E-mail: latex-team@latex-project.org

2 What is a document?

Besides the textual content of the words themselves, the source file of a document contains mark-up elements that add structure to the document. These elements include sectional divisions, figure/table captions, lists of various sorts, theorems/proofs, and so on. The list will be different for every document that can be written.

Each element can be represented logically without worrying about the formatting, with mark-up such as `\section`, `\caption`, `\begin{enumerate}` and so on. The output of each one of these document elements will be a typeset representation of the information marked up, and the visual arrangement and design of these elements can vary widely in producing a variety of desired outcomes.

For each type of document element, there may be design variations that contain the same sort of information but present it in slightly different ways. For example, the difference between a numbered and an unnumbered section, `\section` and `\section*`, or the difference between an itemized list or an enumerated list.

There are three distinct layers in the definition of “a document” at this level

1. semantic elements such as the ideas of sections and lists;
2. a set of design solutions for representing these elements visually;
3. specific variations for these designs that represent the elements in the document.

In the parlance of the template system, these are called types, templates, and instances, and they are discussed below in sections 4, 5, and 7, respectively.

3 Types, templates, and instances

By formally declaring documents to be composed of mark-up elements grouped into types, which are interpreted and typeset with a set of templates, each of which has one or more instances with which to compose each and every semantic unit of the text, we can cleanly separate the components of document construction.

All of the structures provided by the template system are global, and do not respect $\text{\TeX}$ grouping.

4 Template types

An *template type* (sometimes just “type”) is an abstract idea of a document element that takes a fixed number of arguments corresponding to the information from the document author that it is representing. A sectioning type, for example, might take three inputs: “title”, “short title”, and “label”.

Any given document class will define which types are to be used in the document, and any template of a given type can be used to generate an instance for the type. (Of course, different templates will produce different typeset representations, but the underlying content will be the same.)

<code>\NewTemplateType</code>	<code>\NewTemplateType {⟨template type⟩} {⟨no. of args⟩}</code>
-------------------------------	---

This function defines an `⟨template type⟩` taking `⟨number of arguments⟩`, where the `⟨type⟩` is an abstraction as discussed above. For example,

`\NewTemplateType{sectioning}{3}`

creates a type “sectioning”, where each use of that type will need three arguments.

5 Templates

A *template* is a generalized design solution for representing the information of a specified type. Templates that do the same thing, but in different ways, are grouped together by their type and given separate names. There are two important parts to a template:

- the parameters it takes to vary the design it is producing;
- the implementation of the design.

As a document author or designer does not care about the implementation but rather only the interface to the template, these two aspects of the template definition are split into two independent declarations, `\DeclareTemplateInterface` and `\DeclareTemplateCode`.

<code>\DeclareTemplateInterface</code>	<code>\DeclareTemplateInterface</code> <code>{⟨type⟩} {⟨template⟩} {⟨no. of args⟩}</code> <code>{⟨key list⟩}</code>
--	---

A `⟨template⟩` interface is declared for a particular `⟨type⟩`, where the `⟨number of arguments⟩` must agree with the type declaration. The interface itself is defined by the `⟨key list⟩`, which is itself a key–value list taking a specialized format:

`⟨key1⟩ : ⟨key type1⟩ ,`
`⟨key2⟩ : ⟨key type2⟩ ,`
`⟨key3⟩ : ⟨key type3⟩ = ⟨default3⟩ ,`
`⟨key4⟩ : ⟨key type4⟩ = ⟨default4⟩ ,`
`...`

Each `⟨key⟩` name should consist of ASCII characters, with the exception of `,`, `=` and `:`. The recommended form for key names is to use lower case letters, with dashes to separate out different parts. Spaces are ignored in key names, so they can be included or missed out at will. Each `⟨key⟩` must have a `⟨key type⟩`, which defines the type of input that the `⟨key⟩` requires. A full list of key types is given in Table 1. Each key may have a `⟨default⟩` value, which will be used in by the template if the `⟨key⟩` is not set explicitly. The `⟨default⟩` should be of the correct form to be accepted by the `⟨key type⟩` of the `⟨key⟩`: this is not checked by the code. Expressions for numerical values are evaluated when the template is used, thus for example values given in terms of `em` or `ex` will be set respecting the prevailing font.

Key-type	Description of input
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice{⟨choices⟩}</code>	A list of pre-defined <code>⟨choices⟩</code>
<code>commalist</code>	A comma-separated list
<code>function{⟨N⟩}</code>	A (protected) function definition with N arguments (N from 0 to 9)
<code>instance{⟨name⟩}</code>	An instance of type <code>⟨name⟩</code>
<code>integer</code>	An integer or integer expression
<code>length</code>	A fixed length
<code>muskip</code>	A math length with shrink and stretch components
<code>real</code>	A real (floating point) value
<code>skip</code>	A length with shrink and stretch components
<code>tokenlist</code>	A token list: any text or commands

Table 1: Key-types for defining template interfaces with `\DeclareTemplateInterface`.

`\KeyValue` `\KeyValue {⟨key name⟩}`

There are occasions where the default (or value) for one key should be taken from another. The `\KeyValue` function can be used to transfer this information without needing to know the internal implementation of the key:

```

\DeclareTemplateInterface { type } { template } { no. of args }
{
  key-name-1 : key-type = value ,
  key-name-2 : key-type = \KeyValue { key-name-1 },
  ...
}

```

Key-type	Description of binding
<code>boolean</code>	Boolean variable, <i>e.g.</i> <code>\l_tmpa_bool</code>
<code>choice</code>	List of choice implementations (see Section 6)
<code>commalist</code>	Comma list, <i>e.g.</i> <code>\l_tmpa_clist</code>
<code>function</code>	Function taking N arguments, <i>e.g.</i> <code>\use_i:nn</code>
<code>instance</code>	
<code>integer</code>	Integer variable, <i>e.g.</i> <code>\l_tmpa_int</code>
<code>length</code>	Dimension variable, <i>e.g.</i> <code>\l_tmpa_dim</code>
<code>muskip</code>	Muskip variable, <i>e.g.</i> <code>\l_tmpa_muskip</code>
<code>real</code>	Floating-point variable, <i>e.g.</i> <code>\l_tmpa_fp</code>
<code>skip</code>	Skip variable, <i>e.g.</i> <code>\l_tmpa_skip</code>
<code>tokenlist</code>	Token list variable, <i>e.g.</i> <code>\l_tmpa_tl</code>

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Apart from `choice` and `function` all of these accept the keyword `global` to carry out a global assignment. Apart from `choice`, all of these accept the keyword `name` to allow passing command name without the leading backslash: this is useful for auto-generated names.

<code>\DeclareTemplateCode</code>	<code>\DeclareTemplateCode</code>
	<code>{⟨type⟩} {⟨template⟩} {⟨no. of args⟩}</code>
	<code>{⟨key bindings⟩} {⟨code⟩}</code>

The relationship between a templates keys and the internal implementation is created using the `\DeclareTemplateCode` function. As with `\DeclareTemplateInterface`, the `⟨template⟩` name is given along with the `⟨type⟩` and `⟨number of arguments⟩` required. The `⟨key bindings⟩` argument is a key–value list which specifies the relationship between each `⟨key⟩` of the template interface with an underlying `⟨variable⟩`.

```

⟨key1⟩ = ⟨variable1⟩,
⟨key2⟩ = ⟨variable2⟩,
⟨key3⟩ = global ⟨variable3⟩,
⟨key4⟩ = global ⟨variable4⟩,
⟨key5⟩ = name { ⟨name5⟩ },
...
```

As illustrated, the keyword `global` may be included in the listing to indicate that the `⟨variable⟩` should be assigned globally. The keyword `name` may also be used to give the variable by (cs)name rather than as a single token. If both keywords are used, `global` should come before `name`.

A full list of variable bindings is given in Table 2.

The `⟨code⟩` argument of `\DeclareTemplateCode` is used as the replacement text for the template when it is used, either directly or as an instance. This may therefore accept arguments `#1`, `#2`, *etc.* as detailed by the `⟨number of arguments⟩` taken by the type.

<code>\AssignTemplateKeys</code>	<code>\AssignTemplateKeys</code>
----------------------------------	----------------------------------

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template may be delayed by including the command `\AssignTemplateKeys`. If this is *not* present, keys are assigned immediately before the template code. If an `\AssignTemplateKeys` command is present, assignment is delayed until this point. Note that the command must be *directly* present in the code, not placed within a nested command/macro.

<code>\SetKnownTemplateKeys</code>	<code>\SetKnownTemplateKeys</code>	<code>{\type}</code>	<code>{\template}</code>	<code>{\keyvals}</code>
<code>\SetTemplateKeys</code>	<code>\SetTemplateKeys</code>	<code>{\type}</code>	<code>{\template}</code>	<code>{\keyvals}</code>
<code>\SetCurrentKnownTemplateKeys</code>	<code>\SetCurrentKnownTemplateKeys</code>	<code>{\keyvals}</code>		
<code>\UnusedTemplateKeys</code>	<code>\UnusedTemplateKeys</code>	<code>% all \keyvals unused by previous \SetKnownTemplateKeys</code>		

In the final argument of `\DeclareTemplateCode` one can also overwrite (some of) the current template key value settings by using the command `\SetKnownTemplateKeys` or `\SetTemplateKeys`, i.e., they can overwrite the template default values and the values assigned by the instance.

The `\SetKnownTemplateKeys`, `\SetTemplateKeys` and `\SetCurrentKnownTemplateKeys` commands are only supported within the code of a template; using them elsewhere has unpredictable results. If they are used together with `\AssignTemplateKeys` then the latter command should come first in the template code.

The main use case for these commands is the situation where there is an argument (normally `#1`) to the template in which a key/value list can be specified that overwrites the normal settings. In that case one could use

`\SetKnownTemplateKeys{\type}{\template}{\#1}`

to process this key/value list inside the template.

If `\SetKnownTemplateKeys` or `\SetCurrentKnownTemplateKeys` are executed and the `\keyvals` argument contains keys not known to the `\template` they are simply ignored and stored in the tokenlist `\UnusedTemplateKeys` without generating an error. This way it is possible to apply the same key/val list specified by the user on a document-level command or environment to several templates, which is useful, if the command or environment is implemented by calling several different template instances.

`\SetCurrentKnownTemplateKeys` works like `\SetKnownTemplateKeys` but obtains the name of the current `\type`/`\template` automatically based on the current context. It should be used before any other template code may be called (and thus change the value of the tracking of the current template).

As a variation of that, you can use this key/val list the first time, and for the next template instance use what remains in `\UnusedTemplateKeys` (i.e., the key/val list with only the keys that have not been processed previously). The final processing step could then be `\SetTemplateKeys`, which unconditionally attempts to set the `\keyvals` received in its third argument. This command complains if any of them are unknown keys. Alternatively, you could use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty.¹

For example, a list, such as `enumerate`, is made up from a `blockenv`, `block`, `list`,

¹Using `\SetTemplateKeys` exposes the inner structure of the template keys when generating an error. This is something one may want to avoid as it can be confusing to the user, especially if several templates are involved. In that case use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty; if it is not empty then generate your own error message.

and a `para` template and in the single user-supplied optional argument of `enumerate` key/values for any of these templates might be specified.

In fact, in the particular example of list environments, the supplied key/value list is also saved and then applied to each `\item` which is implemented through an `item` template. This way, one can specify one-off settings for all the items of a single list (on the environment level), as well as to individual items within that list (by specifying them in the optional argument of an `\item`). With `\SetKnownTemplateKeys` and `\SetTemplateKeys` working together, it is possible to provide this flexibility and still alert the user when one of their keys is misspelled.

On the other hand you may want to allow for “misspellings” without generating an error or a warning. For example, if you define a template that accepts only a few keys, you might just want to ignore anything specified in the source when you use this template in place of a different one, without the need to alter the document source. Or you might just generate a warning message, which is easy, given that the unused key/values are available in the `\UnusedTemplateKeys` variable.

`\DeclareTemplateCopy`

`\DeclareTemplateCopy`
`{<type>} {<template2>} {<template1>}`

Copies `<template1>` of `<type>` to a new name `<template2>`: the copy can then be edited independent of the original.

6 Multiple choices

The `choice` key type implements multiple choice input. At the interface level, only the list of valid choices is needed:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
{ key-name : choice { A, B, C } }
```

where the choices are given as a comma-list (which must therefore be wrapped in braces). A default value can also be given:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
{ key-name : choice { A, B, C } = A }
```

At the implementation level, each choice is associated with code, using a nested key-value list.

```
\DeclareTemplateCode { foo } { bar } { 0 }
{
  key-name =
  {
    A = Code-A ,
    B = Code-B ,
    C = Code-C
  }
}
{ ... }
```

The two choice lists should match, but in the implementation a special `unknown` choice is also available. This can be used to ignore values and implement an “else” branch:

```

\DeclareTemplateCode { foo } { bar } { 0 }
{
  key-name =
  {
    A      = Code-A ,
    B      = Code-B ,
    C      = Code-C ,
    unknown = Else-code
  }
}
{ ... }

```

The **unknown** entry must be the last one given, and should *not* be listed in the interface part of the template.

For keys which accept the values **true** and **false** both the boolean and choice key types can be used. As template interfaces are intended to prompt clarity at the design level, the boolean key type should be favored, with the choice type reserved for keys which take arbitrary values.

7 Instances

After a template is defined it still needs to be put to use. The parameters that it expects need to be defined before it can be used in a document. Every time a template has parameters given to it, an *instance* is created, and this is the code that ends up in the document to perform the typesetting of whatever pieces of information are input into it.

For example, a template might say “here is a section with or without a number that might be centered or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it” whereas an instance declares “this is a section with a number, which is centered and set in 12pt italic with a 10pt skip before and a 12pt skip after it”. Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

`\DeclareInstance` `\DeclareInstance`

`{\langle type \rangle} {\langle instance \rangle} {\langle template \rangle} {\langle parameters \rangle}`

This function uses a `\langle template \rangle` for an `\langle type \rangle` to create an `\langle instance \rangle`. The `\langle instance \rangle` will be set up using the `\langle parameters \rangle`, which will set some of the `\langle keys \rangle` in the `\langle template \rangle`.

As a practical example, consider a type for document sections (which might include chapters, parts, sections, *etc.*), which is called **sectioning**. One possible template for this type might be called **basic**, and one instance of this template would be a numbered section. The instance declaration might read:

```
\DeclareInstance { sectioning } { section-num } { basic }
{
  numbered      = true ,
  justification = center ,
  font          = \normalsize\itshape ,
  before-skip   = 10pt ,
  after-skip    = 12pt ,
}
```

Of course, the key names here are entirely imaginary, but illustrate the general idea of fixing some settings.

`\InstanceValue` ★ `\InstanceValue {\langle type \rangle} {\langle instance \rangle} {\langle key \rangle}`

Expands to the current value for the `\langle key \rangle` stored in the `\langle instance \rangle` of `\langle type \rangle`. If the `\langle instance \rangle` or `\langle key \rangle` do not exist, the expansion is empty. The result is returned within the `\unexpanded primitive (\exp_not:n)`.

`\IfInstanceExistsT` `\IfInstanceExistsTF {\langle type \rangle} {\langle instance \rangle} {\langle true code \rangle} {\langle false code \rangle}`

`\IfInstanceExistsF` Tests if the named `\langle instance \rangle` of a `\langle type \rangle` exists, and then inserts the appropriate code
`\IfInstanceExistsTF` into the input stream.

`\DeclareInstanceCopy` `\DeclareInstanceCopy`

`{\langle type \rangle} {\langle instance2 \rangle} {\langle instance1 \rangle}`

Copies the `\langle values \rangle` for `\langle instance1 \rangle` for an `\langle type \rangle` to `\langle instance2 \rangle`.

8 Document interface

After the instances have been chosen, document commands must be declared to use those instances in the document. `\UseInstance` calls instances directly, and this command should be used internally in document-level mark-up.

`\UseInstance` `\UseInstance`

`{\langle type \rangle} {\langle instance \rangle} \langle arguments \rangle`

Uses an `\langle instance \rangle` of the `\langle type \rangle`, which will require `\langle arguments \rangle` as determined by the number specified for the `\langle type \rangle`. The `\langle instance \rangle` must have been declared before it can be used, otherwise an error is raised.

<code>\UseTemplate</code>	<code>\UseTemplate {<type>} {<template>}</code> <code>{<settings>} <arguments></code>
---------------------------	--

Uses the `<template>` of the specified `<type>`, applying the `<settings>` and absorbing `<arguments>` as detailed by the `<type>` declaration. This in effect is the same as creating an instance using `\DeclareInstance` and immediately using it with `\UseInstance`, but without the instance having any further existence. This command is therefore useful when a template needs to be used only once.

This function can also be used as the argument to `instance` key types:

```
\DeclareInstance { type } { template } { instance }
{
  instance-key =
    \UseTemplate { type2 } { template2 } { <settings> }
}
```

9 Changing existing definitions

Template parameters may be assigned specific defaults for instances to use if the instance declaration doesn't explicit set those parameters. In some cases, the document designer will wish to edit these defaults to allow them to “cascade” to the instances. The alternative would be to set each parameter identically for each instance declaration, a tedious and error-prone process.

<code>\EditTemplateDefaults</code>	<code>\EditTemplateDefaults</code> <code>{<type>} {<template>} {<new defaults>}</code>
------------------------------------	---

Edits the `<defaults>` for a `<template>` for an `<type>`. The `<new defaults>`, given as a key–value list, replace the existing defaults for the `<template>`. This means that the change will apply to instances declared after the editing, but that instances which have already been created are unaffected.

<code>\EditInstance</code>	<code>\EditInstance</code> <code>{<type>} {<instance>} {<new values>}</code>
----------------------------	---

Edits the `<values>` for an `<instance>` for an `<type>`. The `<new values>`, given as a key–value list, replace the existing values for the `<instance>`. This function is complementary to `\EditTemplateDefaults`: `\EditInstance` changes a single instance while leaving the template untouched.

9.1 Expanding the values of keys

To allow the user to apply expansion of values when the key is set, key names can be followed by an expansion specifier. This is given by appending `:` and a single letter specifier to the key name. These letters are the normal argument specifiers for `expl3`, thus they may be one of `n` (redundant but supported), `o`, `V`, `v`, `e`, `N` (again redundant) or `c`. Expansion of a control sequence name is particularly useful when you need to refer to an internal $\text{\LaTeX} 2_{\epsilon}$ or an L3 programming layer variable, e.g.,

```
key-a:c = @itemdepth , % use \@itemdepth as the value
key-b:v = @itemdepth   % use the current value of \@itemdepth as the value
```

10 Creating equivalent keys

There are times when it is useful to have different names which map to the same underlying key interfaces, for example where a more complex higher level name is implemented by several simpler underlying keys, or where a key name needs to be changed. To support this, it is possible to define equivalent keys using `\DeclareTemplateInterface`.

```
\DeclareTemplateInterfaceEquivalents \DeclareTemplateInterfaceEquivalents
                                   {\langle type \rangle} {\langle template \rangle} [\langle deprecation-date \rangle] {\langle key list \rangle}
```

Defines one or more keys to act as equivalents to the base keys of the `\langle template \rangle`. As for `\DeclareTemplateInterface`, these take a specialized format:

```
\langle key1 \rangle : \langle key type1 \rangle = \langle definition1 \rangle ,
\langle key2 \rangle : \langle key type2 \rangle = \langle definition2 \rangle ,
...
```

The `\langle key type \rangle` of the key being created should be one of

- **alias**: A direct equivalence between the key being created and a single parent
- **meta**: An equivalence to one or more parent keys, where additional code may be included as well as the value passed to the key being created; the value (provided in the instance declaration or in the ad hoc setting in the document) is available as `#1`.

for example

```
\DeclareTemplateInterfaceEquivalents { type } { template }
{
  alias-key-one : alias = base-key-one ,
  alias-key-two : meta =
    { parent-key-two = {#1}, base-key-three = \hspace{#1} }
}
```

Equivalent key names may be reused to overwrite existing equivalent keys but cannot be the same as base keys of the template.

Optionally, the `\langle deprecation date \rangle` may be set. Before this date, a warning will be printed concerning the keys; after this date, an error will be issued. In both cases, the keys will still be defined.

10.1 Modifying existing key values

Some changes to key values are relative adjustments to the previous value, for example to add 1 cm to a margin. To support this, the syntax

```
key-a += 1cm
```

can be used, with a `+` immediately before the `=` sign. This will cause the new value to be *added* to the existing one. This naturally applies to all numeric key types, and for `tokenlist` and `commalist` means adding the new material to the end of the existing value. An error will arise if this is used for other key types, for example `function`.

11 Getting information about templates and instances

	<code>\ShowInstanceValues {<type>} {<instance>}</code>	Shows the <i><values></i> for an <i><instance></i> of the given <i><type></i> at the terminal.
	<code>\ShowTemplateCode {<type>} {<template>}</code>	Shows the <i><code></i> of a <i><template></i> for an <i><type></i> in the terminal.
	<code>\ShowTemplateDefaults {<type>} {<template>}</code>	Shows the <i><default></i> values of a <i><template></i> for an <i><type></i> in the terminal.
	<code>\ShowTemplateInterface {<type>} {<template>}</code>	Shows the <i><keys></i> and associated <i><key types></i> of a <i><template></i> for an <i><type></i> in the terminal.
	<code>\ShowTemplateVariables {<type>} {<template>}</code>	Shows the <i><variables></i> and associated <i><keys></i> of a <i><template></i> for an <i><type></i> in the terminal. Note that <i>code</i> and <i>choice</i> keys do not map directly to variables but to arbitrary code. For <i>choice</i> keys, each valid choice is shown as a separate entry in the list, with the key name and choice separated by a space, for example Template 'example' of type 'example' has variable mapping: > demo unknown => \def \demo {?} > demo c => \def \demo {c} > demo b => \def \demo {b} > demo a => \def \demo {a}.

would be shown for a choice key *demo* with valid choices *a*, *b* and *c*, plus code for an *unknown* branch.

12 Debugging support

	<code>\DebugTemplatesOn</code>	Turn on or off debugging support for use of templates. Debugging information is provided at the point of <i>use</i> of templates and instances, i.e. where normal messages should be avoided for performance reasons.
	<code>\DebugTemplatesOff</code>	

13 The implementation

```
1 <@@=template>
2 <*2kernel>
3 \message{templates,}
4 </2kernel>
```

```

5 <*2ekernel | latexrelease>
6 \ExplSyntaxOn
7 <latexrelease>\NewModuleRelease{2024/06/01}{ltemplates}
8 <latexrelease> {Prototype~document~commands}%

```

13.1 Extensions for expl3

A slight refinement of code from l3doc, likely should move to the core. Note that like other `compare` functions, the arguments are always fully expanded.

```

\__template_date_compare:nNnTF
\__template_date_compare_aux:nNn
\__template_date_compare_auxi:nnnNnnn
\__template_date_compare_auxii:w
\__template_date_compare_auxiii:w

9 \prg_new_conditional:Npnn \__template_date_compare:nNn #1#2#3 { TF }
10 { \use:e { \exp_not:N \__template_date_compare_aux:nNn {#1} #2 {#3} } }
11 \cs_new:Npn \__template_date_compare_aux:nNn #1#2#3
12 {
13   \use:e
14   {
15     \exp_not:N \__template_date_compare_auxi:nnnNnnn
16     \__template_date_compare_auxii:w #1 - - \q_stop
17     #2
18     \__template_date_compare_auxii:w #3 - - \q_stop
19   }
20 }
21 \cs_new:Npn \__template_date_compare_auxi:nnnNnnn #1#2#3#4#5#6#7
22 {
23   \int_compare:nNnTF {#1} = {#5}
24   {
25     \int_compare:nNnTF {#2} = {#6}
26     {
27       \int_compare:nNnTF {#3} #4 {#7}
28       { \prg_return_true: } { \prg_return_false: }
29     }
30     {
31       \int_compare:nNnTF {#2} #4 {#6}
32       { \prg_return_true: } { \prg_return_false: }
33     }
34   }
35   {
36     \int_compare:nNnTF {#1} #4 {#5}
37     { \prg_return_true: } { \prg_return_false: }
38   }
39 }
40 \cs_new:Npn \__template_date_compare_auxii:w #1 - #2 - #3 - #4 \q_stop
41 {
42   \tl_if_blank:nTF {#2#3}
43   { \__template_date_compare_auxiii:w #1 / / / \q_stop }
44   { { 0 #1 } { 0 #2 } { 0 #3 } }
45 }
46 \cs_new:Npn \__template_date_compare_auxiii:w #1 / #2 / #3 / #4 \q_stop
47 { { 0 #1 } { 0 #2 } { 0 #3 } }

```

(End of definition for `\__template_date_compare:nNnTF` and others.)

13.2 Variables and constants

```
\c__template_code_root_tl
\c__template_defaults_root_tl
\c__template_instances_root_tl
\c__template_keytypes_root_tl
\c__template_key_order_root_tl
\c__template_restrict_root_tl
\c__template_values_root_tl
\c__template_vars_root_tl
```

```
\c__template_msg_root_tl
```

So that literal values are kept to a minimum.

```
48 \tl_const:Nn \c__template_code_root_tl { template~code~>~ }
49 \tl_const:Nn \c__template_defaults_root_tl { template~defaults~>~ }
50 \tl_const:Nn \c__template_instances_root_tl { template~instance~>~ }
51 \tl_const:Nn \c__template_keytypes_root_tl { template~key~types~>~ }
52 \tl_const:Nn \c__template_key_order_root_tl { template~key~order~>~ }
53 \tl_const:Nn \c__template_msg_root_tl { template~message~>~ }
54 \tl_const:Nn \c__template_values_root_tl { template~values~>~ }
55 \tl_const:Nn \c__template_vars_root_tl { template~vars~>~ }
```

```
\c__template_keytypes_seq
```

A list of all keytypes.

```
56 \exp_args:Nne \seq_const_from_clist:Nn \c__template_keytypes_seq
57 {
58   \tl_to_str:n
59   {
60     boolean ,
61     choice ,
62     commalist ,
63     function ,
64     instance ,
65     integer ,
66     length ,
67     muskip ,
68     real ,
69     skip ,
70     tokenlist
71   }
72 }
```

```
\c__template_keytypes_arg_seq
```

A list of keytypes which also need additional data (an argument), used to parse the keytype correctly.

```
73 \seq_const_from_clist:Nn \c__template_keytypes_arg_seq
74 { choice , function , instance }
```

```
\g__template_type_prop
```

For storing types and the associated number of arguments.

```
75 \prop_new:N \g__template_type_prop
```

\l__template_assignments_tl

When creating an instance, the assigned values are collected here.

```
76 \tl_new:N \l__template_assignments_tl
```

\l__template_default_tl

The default value for a key is recovered here from the property list in which it is stored.

```
77 \tl_new:N \l__template_default_tl
```

\l__template_error_bool

A flag for errors to be carried forward.

```
78 \bool_new:N \l__template_error_bool
```

\l__template_global_bool

Used to indicate that assignments should be global.

```
79 \bool_new:N \l__template_global_bool
```

\l__template_key_name_tl
\l__template_keytype_tl
\l__template_keytype_arg_tl
\l__template_value_tl
\l__template_var_tl

When defining each key in a template, the name and type of the key need to be separated and stored. Any argument needed by the keytype is also stored separately.

```
80 \tl_new:N \l__template_key_name_tl
81 \tl_new:N \l__template_keytype_tl
82 \tl_new:N \l__template_keytype_arg_tl
83 \tl_new:N \l__template_value_tl
84 \tl_new:N \l__template_var_tl
```

\l__template_value_exp_str

```
85 \str_new:N \l__template_value_exp_str
```

\l__template_keytypes_prop

\l__template_key_order_seq

\l__template_value_order_seq

\l__template_vars_prop

To avoid needing too many difficult-to-follow csname assignments, various scratch token registers are used to build up data, which is then transferred

```
86 \prop_new:N \l__template_keytypes_prop
87 \seq_new:N \l__template_key_order_seq
88 \prop_new:N \l__template_values_prop
89 \seq_new:N \l__template_value_order_seq
90 \prop_new:N \l__template_vars_prop
```

\l__template_msg_prop

```
91 \prop_new:N \l__template_msg_prop
```

<code>\l__template_tmp_clist</code>	Scratch space.
<code>\l__template_tmp_dim</code>	
<code>\l__template_tmp_int</code>	92 <code>\clist_new:N \l__template_tmp_clist</code>
<code>\l__template_tmp_muskip</code>	93 <code>\dim_new:N \l__template_tmp_dim</code>
<code>\l__template_tmp_skip</code>	94 <code>\int_new:N \l__template_tmp_int</code>
<code>\l__template_tmp_tl</code>	95 <code>\muskip_new:N \l__template_tmp_muskip</code>
	96 <code>\skip_new:N \l__template_tmp_skip</code>
	97 <code>\tl_new:N \l__template_tmp_tl</code>

<code>\l__template_current_type_tl</code>	
<code>\l__template_current_template_tl</code>	

For setting the current template values.

```

98 \tl_new:N \l__template_current_type_tl
99 \tl_new:N \l__template_current_template_tl

```

`\__template_tmp:n` Scratch space.

```
100 \cs_new:Npn \__template_tmp:n #1 { }
```

(End of definition for __template_tmp:n.)

`\s__template_mark` Internal scan marks.

```

101 \scan_new:N \s__template_mark
102 \scan_new:N \s__template_stop

```

`\q__template_nil` Internal quarks.

```
103 \quark_new:N \q__template_nil
```

`\__template_quark_if_nil_p:n` Branching quark conditional.

`\__template_quark_if_nil:nTF` 104 `\__kernel_quark_new_conditional:Nn \__template_quark_if_nil:N { F }`

(End of definition for __template_quark_if_nil:nTF.)

13.3 Debugging support

<code>__template_debug_on:</code>	
<code>__template_debug_off:</code>	

```

105 \cs_new_protected:Npn \__template_debug_on:
106 {
107   \cs_set_protected:Npn \__template_debug:n ##1 {##1}
108   \cs_set_protected:Npn \__template_debug_typeof:n ##1
109     { \iow_term:e { [ Template ] ~ ==> ~ ##1 } }
110 }
111 \cs_new_protected:Npn \__template_debug_off:
112 {
113   \cs_set_protected:Npn \__template_debug:n ##1 { }
114   \cs_set_protected:Npn \__template_debug_typeof:n ##1 { }
115 }

```

(End of definition for __template_debug_on: and __template_debug_off:.)

```

    \__template_debug:n
\__template_debug_typeout:n
116 \cs_new_protected:Npn \__template_debug:n #1 { }
117 \cs_new_protected:Npn \__template_debug_typeout:n #1 { }

(End of definition for \__template_debug:n and \__template_debug_typeout:n.)

```

13.4 Testing existence and validity

There are a number of checks needed for either the existence of a type, template or instance. There are also some for the validity of a particular call. All of these are collected up here.

__template_execute_if_arg_agree:nnT A test agreement between the number of arguments for the template type and that specified when creating a template. This is not done as a separate conditional for efficiency and better error message

```

118 \cs_new_protected:Npn \__template_execute_if_arg_agree:nnT #1#2#3
119 {
120     \prop_get:NnN \g__template_type_prop {#1} \l__template_tmp_tl
121     \int_compare:nNnTF {#2} = \l__template_tmp_tl
122         {#3}
123         {
124             \msg_error:nneee { template } { argument-number-mismatch }
125             {#1} { \l__template_tmp_tl } {#2}
126         }
127 }

```

(End of definition for __template_execute_if_arg_agree:nnT.)

__template_execute_if_code_exist:nnT A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.

```

128 \cs_new_protected:Npn \__template_execute_if_code_exist:nnT #1#2#3
129 {
130     \cs_if_exist:cTF { \c__template_code_root_tl #1 / #2 }
131         {#3}
132         { \msg_error:nnnn { template } { no-template-code } {#1} {#2} }
133 }

```

(End of definition for __template_execute_if_code_exist:nnT.)

__template_execute_if_keytype_exist:nT
__template_execute_if_keytype_exist:VT The test for valid keytypes looks for a function to set up the key, which is part of the “code” side of the template definition. This avoids having different lists for the two parts of the process.

```

134 \cs_new_protected:Npn \__template_execute_if_keytype_exist:nT #1#2
135 {
136     \seq_if_in:NeTF \c__template_keytypes_seq { \tl_to_str:n {#1} }
137         {#2}
138         { \msg_error:nnn { template } { unknown-keytype } {#1} }
139 }
140 \cs_generate_variant:Nn \__template_execute_if_keytype_exist:nT { V }

```

(End of definition for __template_execute_if_keytype_exist:nT.)

_template_execute_if_type_exist:nT To check that a particular type is valid.

```

141 \cs_new_protected:Npn \_template_execute_if_type_exist:nT #1#2
142 {
143   \prop_if_in:NnTF \g__template_type_prop {#1}
144     {#2}
145     { \msg_error:nnn { template } { unknown-type } {#1} }
146 }

```

(End of definition for _template_execute_if_type_exist:nT.)

_template_execute_if_keyname_free:nT

_template_execute_if_keyname_free:VT

```

147 \cs_new_protected:Npn \_template_execute_if_keyname_free:nT #1#2
148 {
149   \seq_if_in:NnTF \l__template_key_order_seq {#1}
150     { \msg_error:nnn { template } { duplicate-key-interface } {#1} }
151     {#2}
152 }
153 \cs_generate_variant:Nn \_template_execute_if_keyname_free:nT { V }

```

(End of definition for _template_execute_if_keyname_free:nT.)

_template_execute_if_keys_exist:nnT

To check that the keys for a template have been set up before trying to create any code, a simple check for the correctly-named keytype property list.

```

154 \cs_new_protected:Npn \_template_if_keys_exist:nnT #1#2#3
155 {
156   \cs_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
157     {#3}
158     { \msg_error:nnnn { template } { unknown-template } {#1} {#2} }
159 }

```

(End of definition for _template_execute_if_keys_exist:nnT.)

_template_if_key_value:nTF

Tests for the first token in a string being \KeyValue.

_template_if_key_value:VTF

```

160 \prg_new_conditional:Nppn \_template_if_key_value:n #1 { T , F , TF }
161 {
162   \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil \q_stop }
163   \prg_return_true:
164   \prg_return_false:
165 }
166 \prg_generate_conditional_variant:Nnn \_template_if_key_value:n { V } { T , F , TF }

```

(End of definition for _template_if_key_value:nTF.)

_template_if_instance_exist:nnTF

Testing for an instance

```

167 \prg_new_conditional:Nppn \_template_if_instance_exist:nn #1#2 { T , F , TF }
168 {
169   \cs_if_exist:cTF { \c__template_instances_root_tl #1 / #2 }
170   \prg_return_true:
171   \prg_return_false:
172 }

```

(End of definition for _template_if_instance_exist:nnTF.)

```

\__template_if_use_template:nTF Tests for the first token in a string being \UseTemplate.
173 \prg_new_conditional:Npnn \__template_if_use_template:n #1 { TF }
174 {
175     \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
176     \prg_return_true:
177     \prg_return_false:
178 }

```

(End of definition for __template_if_use_template:nTF.)

13.5 Saving and recovering property lists

The various property lists for templates have to be shuffled in and out of storage.

```

\__template_store_defaults:nn The defaults and keytypes are transferred from the scratch property lists to the “proper”
\__template_store_keytypes:nn lists for the template being created.
\__template_store_values:nn
\__template_store_vars:nn
179 \cs_new_protected:Npn \__template_store_defaults:nn #1#2
180 {
181     \debug_suspend:
182     \prop_gclear_new:c { \c__template_defaults_root_tl #1 / #2 }
183     \prop_gset_eq:cN { \c__template_defaults_root_tl #1 / #2 }
184     \l__template_values_prop
185     \debug_resume:
186 }
187 \cs_new_protected:Npn \__template_store_keytypes:nn #1#2
188 {
189     \debug_suspend:
190     \prop_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
191     {
192         \msg_info:nnnn { template } { declare-template-interface } {#1} {#2}
193         \prop_gclear:c { \c__template_keytypes_root_tl #1 / #2 }
194     }
195     { \prop_new:c { \c__template_keytypes_root_tl #1 / #2 } }
196     \prop_gset_eq:cN { \c__template_keytypes_root_tl #1 / #2 }
197     \l__template_keytypes_prop
198     \seq_gclear_new:c { \c__template_key_order_root_tl #1 / #2 }
199     \seq_gset_eq:cN { \c__template_key_order_root_tl #1 / #2 }
200     \l__template_key_order_seq
201     \debug_resume:
202 }
203 \cs_new_protected:Npn \__template_store_values:nn #1#2
204 {
205     \debug_suspend:
206     \prop_gclear_new:c { \c__template_values_root_tl #1 / #2 }
207     \prop_gset_eq:cN { \c__template_values_root_tl #1 / #2 }
208     \l__template_values_prop
209     \debug_resume:
210 }
211 \cs_new_protected:Npn \__template_store_vars:nn #1#2
212 {
213     \debug_suspend:
214     \prop_gclear_new:c { \c__template_msg_root_tl #1 / #2 }
215     \prop_gset_eq:cN { \c__template_msg_root_tl #1 / #2 }
216     \l__template_msg_prop

```

```

217     \prop_gclear_new:c { \c__template_vars_root_tl #1 / #2 }
218     \prop_gset_eq:cN { \c__template_vars_root_tl #1 / #2 }
219         \l__template_vars_prop
220     \debug_resume:
221 }

```

(End of definition for __template_store_defaults:nn and others.)

```

\__template_recover_defaults:nn
\__template_recover_keytypes:nn
\__template_recover_values:nn
\__template_recover_vars:nn

```

Recovering the stored data for a template is rather less complex than storing it. All that happens is the data is transferred from the permanent to the scratch storage. However, we need to check the scratch storage does exist.

```

222 \cs_new_protected:Npn \__template_recover_defaults:nn #1#2
223 {
224     \prop_if_exist:cTF
225     { \c__template_defaults_root_tl #1 / #2 }
226     {
227         \prop_set_eq:Nc \l__template_values_prop
228         { \c__template_defaults_root_tl #1 / #2 }
229     }
230     { \prop_clear:N \l__template_values_prop }
231 }
232 \cs_new_protected:Npn \__template_recover_keytypes:nn #1#2
233 {
234     \prop_if_exist:cTF
235     { \c__template_keytypes_root_tl #1 / #2 }
236     {
237         \prop_set_eq:Nc \l__template_keytypes_prop
238         { \c__template_keytypes_root_tl #1 / #2 }
239     }
240     { \prop_clear:N \l__template_keytypes_prop }
241     \seq_if_exist:cTF { \c__template_key_order_root_tl #1 / #2 }
242     {
243         \seq_set_eq:Nc \l__template_key_order_seq
244         { \c__template_key_order_root_tl #1 / #2 }
245     }
246     { \seq_clear:N \l__template_key_order_seq }
247 }
248 \cs_new_protected:Npn \__template_recover_values:nn #1#2
249 {
250     \prop_if_exist:cTF
251     { \c__template_values_root_tl #1 / #2 }
252     {
253         \prop_set_eq:Nc \l__template_values_prop
254         { \c__template_values_root_tl #1 / #2 }
255     }
256     { \prop_clear:N \l__template_values_prop }
257 }
258 \cs_new_protected:Npn \__template_recover_vars:nn #1#2
259 {
260     \prop_if_exist:cTF
261     { \c__template_msg_root_tl #1 / #2 }
262     {
263         \prop_set_eq:Nc \l__template_msg_prop
264         { \c__template_msg_root_tl #1 / #2 }

```

```

265     }
266     { \prop_clear:N \l__template_msg_prop }
267 \prop_if_exist:cTF
268 { \c__template_vars_root_tl #1 / #2 }
269 {
270     \prop_set_eq:Nc \l__template_vars_prop
271     { \c__template_vars_root_tl #1 / #2 }
272 }
273 { \prop_clear:N \l__template_vars_prop }
274 }

```

(End of definition for `\__template_recover_defaults:nn` and others.)

13.6 Creating new template types

`\__template_define_type:nn` Although the type is the “top level” of the template system, it is actually very easy to implement. All that happens is that the number of arguments required is recorded, indexed by the name of the type.

```

275 \cs_new_protected:Npn \__template_define_type:nn #1#2
276 {
277     \prop_if_in:NnTF \g__template_type_prop {#1}
278     { \msg_error:nnn { template } { type-already-defined } {#1} }
279     { \__template_declare_type:nn {#1} {#2} }
280 }
281 \cs_new_protected:Npn \__template_declare_type:nn #1#2
282 {
283     \int_set:Nn \l__template_tmp_int {#2}
284     \int_compare:nTF { 0 <= \l__template_tmp_int <= 9 }
285     {
286         \msg_info:nnnV { template } { declare-type }
287         {#1} \l__template_tmp_int
288         \prop_gput:NnV \g__template_type_prop {#1}
289         \l__template_tmp_int
290     }
291     {
292         \msg_error:nnnV { template } { bad-number-of-arguments }
293         {#1} \l__template_tmp_int
294     }
295 }

```

(End of definition for `\__template_define_type:nn` and `\__template_declare_type:nn`.)

13.7 Design part of template declaration

The “design” part of a template declaration defines the general behaviour of each key, and possibly a default value. However, it does not include the implementation. This means that what happens here is the two properties are saved to appropriate lists, which can then be used later to recover the information when implementing the keys.

`\__template_declare_template_keys:nnnn` The main function for the “design” part of creating a template starts by checking that the type exists and that the number of arguments required agree. If that is all fine, then the two storage areas for defaults and keytypes are initialized. The mechanism is then

set up for the l3keys module to actually parse the keys. Finally, the code hands of to the storage routine to save the parsed information properly.

```

296 \cs_new_protected:Npn \__template_declare_template_keys:nnnn #1#2#3#4
297 {
298   \__template_execute_if_type_exist:nT {#1}
299   {
300     \__template_execute_if_arg_agree:nnT {#1} {#3}
301     {
302       \prop_clear:N \l__template_values_prop
303       \prop_clear:N \l__template_keytypes_prop
304       \prop_clear:N \l__template_vars_prop
305       \seq_clear:N \l__template_key_order_seq
306       \keyval_parse:NNn
307       \__template_parse_keys_elt:n \__template_parse_keys_elt:nn {#4}
308       \__template_store_defaults:nn {#1} {#2}
309       \__template_store_keytypes:nn {#1} {#2}
310       \__template_store_vars:nn {#1} {#2}
311     }
312   }
313 }

```

(End of definition for __template_declare_template_keys:nnnn.)

__template_parse_keys_elt:n Processing the key part of the key-value pair is always carried out using this function, even if a value was found. First, the key name is separated from the keytype, and if necessary the keytype is separated into two parts. This information is then used to check that the keytype is valid, before storing the keytype (plus argument if necessary) as a property of the key name. The key name is also stored (in braces) in the token list to record the order the keys are defined in.

```

314 \cs_new_protected:Npn \__template_parse_keys_elt:n #1
315 {
316   \__template_split_keytype:n {#1}
317   \bool_if:NF \l__template_error_bool
318   {
319     \__template_execute_if_keytype_exist:VT \l__template_keytype_tl
320     {
321       \seq_map_function:NN \c__template_keytypes_arg_seq
322       \__template_parse_keys_elt_aux:n
323       \bool_if:NF \l__template_error_bool
324       {
325         \__template_execute_if_keyname_free:VT
326         \l__template_key_name_tl
327         { \__template_parse_keys_elt_aux: }
328       }
329     }
330   }
331 }
332 \cs_new_protected:Npn \__template_parse_keys_elt_aux:n #1
333 {
334   \str_if_eq:VnT \l__template_keytype_tl {#1}
335   {
336     \tl_if_empty:NT \l__template_keytype_arg_tl
337     {
338       \msg_error:nnn { template } { keytype-requires-argument } {#1}

```

```

339         \bool_set_true:N \l__template_error_bool
340         \seq_map_break:
341     }
342 }
343 }
344 \cs_new_protected:Npn \__template_parse_keys_elt_aux:
345 {
346     \tl_set:Nx \l__template_tmp_tl
347     {
348         \l__template_keytype_tl
349         \tl_if_empty:NF \l__template_keytype_arg_tl
350         { { \l__template_keytype_arg_tl } }
351     }
352     \prop_put:NVV \l__template_keytypes_prop \l__template_key_name_tl
353     \l__template_tmp_tl
354     \seq_put_right:NV \l__template_key_order_seq \l__template_key_name_tl
355     \str_if_eq:VnT \l__template_keytype_tl { choice }
356     {
357         \clist_if_in:NnT \l__template_keytype_arg_tl { unknown }
358         { \msg_error:nn { template } { choice-unknown-reserved } }
359     }
360 }

```

(End of definition for __template_parse_keys_elt:n, __template_parse_keys_elt_aux:n, and __template_parse_keys_elt_aux:.)

__template_parse_keys_elt:nn For keys which have a default, the keytype and key name are first separated out by the __template_parse_keys_elt:n routine, before storing the default value in the scratch property list.

```

361 \cs_new_protected:Npn \__template_parse_keys_elt:nn #1#2
362 {
363     \__template_parse_keys_elt:n {#1}
364     \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#2}
365 }

```

(End of definition for __template_parse_keys_elt:nn.)

__template_split_keytype:n The keytype and key name should be separated by :. As the definition might be given inside or outside of a code block, the category code of colons is standardized. After that, the standard delimited argument method is used to separate the two parts.

```

366 \cs_new_protected:Npe \__template_split_keytype:n #1
367 {
368     \exp_not:N \bool_set_false:N \exp_not:N \l__template_error_bool
369     \tl_set:Nn \exp_not:N \l__template_tmp_tl {#1}
370     \tl_replace_all:Nnn \exp_not:N \l__template_tmp_tl { : } { \token_to_str:N : }
371     \tl_if_in:VnTF \exp_not:N \l__template_tmp_tl { \token_to_str:N : }
372     {
373         \exp_not:n
374         {
375             \tl_clear:N \l__template_key_name_tl
376             \exp_after:wN \__template_split_keytype_aux:w
377             \l__template_tmp_tl \s__template_stop
378         }
379     }

```

```

380     {
381         \exp_not:N \bool_set_true:N \exp_not:N \l__template_error_bool
382         \msg_error:nnn { template } { missing-keytype } {#1}
383     }
384 }
385 \use:e
386 {
387     \cs_new_protected:Npn \exp_not:N \__template_split_keytype_aux:w
388     #1 \token_to_str:N : #2 \s__template_stop
389     {
390         \tl_put_right:Ne \exp_not:N \l__template_key_name_tl
391         {
392             \exp_not:N \tl_trim_spaces:e
393             { \exp_not:N \tl_to_str:n {#1} }
394         }
395         \tl_if_in:nnTF {#2} { \token_to_str:N : }
396         {
397             \tl_put_right:Nn \exp_not:N \l__template_key_name_tl
398             { \token_to_str:N : }
399             \exp_not:N \__template_split_keytype_aux:w #2 \s__template_stop
400         }
401         {
402             \exp_not:N \tl_if_empty:NTF \exp_not:N \l__template_key_name_tl
403             {
404                 \msg_error:nnn { template } { empty-key-name }
405                 { \token_to_str:N : #2 }
406             }
407             { \exp_not:N \__template_split_keytype_arg:n {#2} }
408         }
409     }
410 }

```

(End of definition for `\__template_split_keytype:n` and `\__template_split_keytype_aux:w`.)

```

\__template_split_keytype_arg:n
\__template_split_keytype_arg:V
\__template_split_keytype_arg_aux:n
\__template_split_keytype_arg_aux:w

```

The second stage of sorting out the keytype is to check for an argument. As there is no convenient delimiting token to look for, a check is made instead for each possible text value for the keytype. To keep things faster, this only involves the keytypes that need an argument. If a match is made, then a check is also needed to see that it is at the start of the keytype information. All being well, the split can then be applied. Any non-matching keytypes are assumed to be “correct” as given, and are left alone (this is checked by other code).

```

411 \cs_new_protected:Npn \__template_split_keytype_arg:n #1
412 {
413     \tl_set:Ne \l__template_keytype_tl { \tl_trim_spaces:n {#1} }
414     \tl_clear:N \l__template_keytype_arg_tl
415     \cs_set_protected:Npn \__template_split_keytype_arg_aux:n ##1
416     {
417         \tl_if_in:nnT {#1} {##1}
418         {
419             \cs_set:Npn \__template_split_keytype_arg_aux:w
420             ####1 ##1 ####2 \s__template_stop
421             {
422                 \tl_if_blank:nT {####1}
423                 {

```

```

424         \tl_set:Nc \l__template_keytype_tl
425         { \tl_trim_spaces:n {##1} }
426         \tl_if_blank:nF {####2}
427         {
428             \tl_set:Nc \l__template_keytype_arg_tl
429             { \use:n ####2 }
430         }
431         \seq_map_break:
432     }
433 }
434 \__template_split_keytype_arg_aux:w #1 \s__template_stop
435 }
436 }
437 \seq_map_function:NN \c__template_keytypes_arg_seq
438 \__template_split_keytype_arg_aux:n
439 }
440 \cs_generate_variant:Nn \__template_split_keytype_arg:n { V }
441 \cs_new:Npn \__template_split_keytype_arg_aux:n #1 { }
442 \cs_new:Npn \__template_split_keytype_arg_aux:w #1 \s__template_stop { }

```

(End of definition for `\__template_split_keytype_arg:n`, `\__template_split_keytype_arg_aux:n`, and `\__template_split_keytype_arg_aux:w`.)

13.8 Equivalent interfaces

`\__template_declare_template_equiv:nnnn`

```

443 \cs_new_protected:Npn \__template_declare_template_equiv:nnnn #1#2#3#4
444 {
445     \__template_execute_if_type_exist:nT {#1}
446     {
447         \__template_if_keys_exist:nnT {#1} {#2}
448         {
449             \__template_recover_keytypes:nn {#1} {#2}
450             \__template_recover_vars:nn {#1} {#2}
451             \keyval_parse:nnn
452             { \__template_parse_equiv_elt:n }
453             { \__template_parse_equiv_elt:nnnn { #1 / #2 } {#3} }
454             {#4}
455             \__template_store_keytypes:nn {#1} {#2}
456             \__template_store_vars:nn {#1} {#2}
457         }
458     }
459 }

```

(End of definition for `\__template_declare_template_equiv:nnnn`.)

`\__template_parse_equiv_elt:n`

We must have key(s) to map to.

```

460 \cs_new_protected:Npn \__template_parse_equiv_elt:n #1
461 { \msg_error:nnn { template } { key-no-equiv } {#1} }

```

(End of definition for `\__template_parse_equiv_elt:n`.)

`\__template_parse_equiv_elt:nnnn`
`\__template_parse_equiv_elt_aux:nnn`
`\__template_parse_equiv_alias:nnnn`
`\__template_parse_equiv_meta:nnnn`
`\__template_parse_equiv_auxi:nnnn`
`\__template_parse_equiv_auxii:n`
`\__template_parse_equiv_auxiii:nnnn`

Most of the work is deferred here: the only thing done is to sanity-check the key names to avoid duplication.

```

462 \cs_new_protected:Npn \__template_parse_equiv_elt:nnnn #1#2#3#4
463 {
464   \__template_split_keytype:n {#3}
465   \bool_if:NF \l__template_error_bool
466   {
467     \prop_get:NVNTF \l__template_keytypes_prop
468     \l__template_key_name_tl \l__template_tmp_tl
469     {
470       \seq_if_in:NeTF \c__template_keytypes_seq
471       { \tl_to_str:N \l__template_tmp_tl }
472       {
473         \msg_error:nnV { template } { equiv-key-interface }
474         \l__template_key_name_tl
475       }
476       { \__template_parse_equiv_elt_aux:nnn {#1} {#2} {#4} }
477     }
478     { \__template_parse_equiv_elt_aux:nnn {#1} {#2} {#4} }
479   }
480 }
481 \cs_new_protected:Npn \__template_parse_equiv_elt_aux:nnn #1#2#3
482 {
483   \cs_if_exist_use:cF
484   { __template_parse_equiv_ \l__template_keytype_tl :nnn }
485   {
486     \msg_error:nnV { template } { unknown-equiv-keytype }
487     \l__template_keytype_tl
488     \use_none:nnn
489   }
490   {#1} {#2} {#3}
491 }

```

As we only need the key name here, we use an inline mapping and the same code for aliases and meta keys. Note the need to detokenize key names for the sequence check.

```

492 \cs_new_protected:Npn \__template_parse_equiv_alias:nnn #1#2#3
493 { \__template_parse_equiv_auxi:nnnn {#1} {#2} { { #3 = {##1} } {#3} } }
494 \cs_new_protected:Npn \__template_parse_equiv_meta:nnn #1#2#3
495 { \__template_parse_equiv_auxi:nnnn {#1} {#2} {#3} {#3} }
496 \cs_new_protected:Npn \__template_parse_equiv_auxi:nnnn #1#2#3#4
497 {
498   \keyval_map_inline:nnn {#4}
499   { \__template_parse_equiv_auxii:n {##1} }
500   { \__template_parse_equiv_auxii:n {##1} }
501   \__template_parse_equiv_auxiii:nnnn {#1} {#2} {#3} {#4}
502 }
503 \cs_new_protected:Npn \__template_parse_equiv_auxii:n #1
504 {
505   \seq_if_in:NeF \l__template_key_order_seq { \tl_to_str:n {#1} }
506   {
507     \msg_error:nnn { template } { unknown-key } {#1}
508     \keyval_map_break:n { \use_none:nn }
509   }
510 }
511 \cs_new_protected:Npn \__template_parse_equiv_auxiii:nnnn #1#2#3#4
512 {

```

```

513     \prop_put:NVV \l__template_keytypes_prop \l__template_key_name_tl
514     \l__template_keytype_tl
515     \seq_put_right:NV \l__template_key_order_seq \l__template_key_name_tl
516     \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl {#4}
517     \tl_if_empty:nF {#2}
518     {
519         \prop_put:NVe \l__template_msg_prop \l__template_key_name_tl
520         { \exp_not:n {#2} { \exp_not:V \l__template_key_name_tl } }
521     }
522 }

```

(End of definition for `\__template_parse_equiv_elt:nnnn` and others.)

13.9 Implementation part of template declaration

`\__template_declare_template_code:nnnnn`
`\__template_declare_template_code:nnnn`

The main function for implementing a template starts with a couple of simple checks to make sure that there are no obvious mistakes: the number of arguments must agree and the template keys must have been declared.

```

523 \cs_new_protected:Npn \__template_declare_template_code:nnnnn #1#2#3#4#5
524 {
525     \__template_execute_if_type_exist:nT {#1}
526     {
527         \__template_execute_if_arg_agree:nnT {#1} {#3}
528         {
529             \__template_if_keys_exist:nnT {#1} {#2}
530             {
531                 \__template_store_key_implementation:nnn {#1} {#2} {#4}
532                 \str_if_in:nnTF {#5} { AssignTemplateKeys }
533                 {
534                     \regex_match:nnTF { \c { AssignTemplateKeys } } {#5}
535                     { \__template_declare_template_code:nnnn {#1} {#2} {#3} {#5} }
536                     {
537                         \__template_declare_template_code:nnnn
538                         {#1} {#2} {#3} { \AssignTemplateKeys #5 }
539                     }
540                 }
541                 {
542                     \__template_declare_template_code:nnnn
543                     {#1} {#2} {#3} { \AssignTemplateKeys #5 }
544                 }
545             }
546         }
547     }
548 }
549 \cs_new_protected:Npn \__template_declare_template_code:nnnn #1#2#3#4
550 {
551     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
552     { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
553     \cs_generate_from_arg_count:cNnn
554     { \c__template_code_root_tl #1 / #2 }
555     \cs_gset_protected:Npn {#3} {#4}
556 }

```

(End of definition for `\__template_declare_template_code:nnnnn` and `\__template_declare_template_code:nnnn`.)

_template_store_key_implementation:nnn

Actually storing the implementation part of a template is quite easy as it only requires the list of keys given to be turned into a property list. There is also some error-checking to do, hence the need to have the list of defined keytypes available. In certain cases (when choices are involved) parsing the key results in changes to the default values. That is why they are loaded and then saved again.

```
557 \\cs_new_protected:Npn \\_template_store_key_implementation:nnn #1#2#3
558 {
559   \\_template_recover_defaults:nn {#1} {#2}
560   \\_template_recover_keytypes:nn {#1} {#2}
561   \\_template_recover_vars:nn {#1} {#2}
562   \\keyval_parse:nnn
563   { \\_template_parse_vars_elt:n } { \\_template_parse_vars_elt:nnn { #1 / #2 } } {#3}
564   \\_template_store_vars:nn {#1} {#2}
565   \\prop_map_inline:Nn \\l__template_keytypes_prop
566   {
567     \\bool_lazy_or:nnF
568     { \\str_if_eq_p:nn {##2} { alias } }
569     { \\str_if_eq_p:nn {##2} { meta } }
570     {
571       \\msg_error:nnnnn { template } { key-not-implemented }
572       {##1} {#2} {#1}
573     }
574   }
575 }
```

(End of definition for _template_store_key_implementation:nnn.)

_template_parse_vars_elt:n

At the implementation stage, every key must have a value given. So this is an error function.

```
576 \\cs_new_protected:Npn \\_template_parse_vars_elt:n #1
577 { \\msg_error:nnn { template } { key-no-variable } {#1} }
```

(End of definition for _template_parse_vars_elt:n.)

_template_parse_vars_elt:nnn
_template_parse_vars_elt_auxi:nn
_template_parse_vars_elt_auxii:nw
_template_parse_vars_elt_auxiii:nnn
_template_parse_vars_elt_auxiii:enn
_template_parse_vars_elt_auxiv:nnw
_template_parse_vars_elt_auxv:nnn
_template_parse_vars_elt_auxv:enn

The actual storage part here is very simple: the storage bin name is placed into the property list. At the same time, a comparison is made with the keytypes defined earlier: if there is a mismatch then an error is raised.

```
578 \\cs_new_protected:Npn \\_template_parse_vars_elt:nnn #1#2#3
579 {
580   \\tl_set:Ne \\l__template_key_name_tl
581   { \\tl_trim_spaces:e { \\tl_to_str:n {#2} } }
582   \\prop_get:NVNTF \\l__template_keytypes_prop
583   \\l__template_key_name_tl
584   \\l__template_keytype_tl
585   {
586     \\_template_split_keytype_arg:V \\l__template_keytype_tl
587     \\_template_parse_vars_elt_auxi:nn {#1} {#3}
588     \\prop_remove:NV \\l__template_keytypes_prop \\l__template_key_name_tl
589   }
590   { \\msg_error:nnn { template } { unknown-key } {#2} }
591 }
```

Split off any leading global and they look for the way to implement.

```

592 \cs_new_protected:Npn \__template_parse_vars_elt_auxi:nn #1#2
593 {
594   \__template_parse_vars_elt_auxii:nw {#1} #2 global global \s__template_stop
595 }
596 \cs_new_protected:Npn \__template_parse_vars_elt_auxii:nw
597 #1#2 global #3 global #4 \s__template_stop
598 {
599   \tl_if_blank:nTF {#4}
600   { \__template_parse_vars_elt_auxiii:nnn {#2} {#1} { } }
601   {
602     \tl_if_blank:nTF {#2}
603     {
604       \__template_parse_vars_elt_auxiii:enn
605       { \tl_trim_spaces:n {#3} }
606       {#1} { global }
607     }
608     { \msg_error:nnn { template } { bad-variable } { #2 global #3 } }
609   }
610 }
611 \cs_new_protected:Npn \__template_parse_vars_elt_auxiii:nnn #1#2#3
612 {
613   \cs_if_exist:cTF { __template_parse_vars_ \l__template_keytype_tl :nnn }
614   {
615     \use:c { __template_parse_vars_ \l__template_keytype_tl :nnn }
616     {#1} {#2} {#3}
617   }
618   {
619     \__template_parse_vars_elt_auxiv:nnw
620     {#2} {#3} #1 name name \s__template_stop
621   }
622 }
623 \cs_generate_variant:Nn \__template_parse_vars_elt_auxiii:nnn { e }
624 \cs_new_protected:Npn \__template_parse_vars_elt_auxiv:nnw
625 #1#2#3 name #4 name #5 \s__template_stop
626 {
627   \tl_if_blank:nTF {#5}
628   { \__template_parse_vars_elt_auxv:nnn {#3} {#1} {#2} }
629   {
630     \tl_if_blank:nTF {#3}
631     {
632       \__template_parse_vars_elt_auxv:enn
633       { \exp_not:c { \tl_trim_spaces:n {#4} } } {#1} {#2}
634     }
635     { \msg_error:nnn { template } { bad-variable } { #3 name #4 } }
636   }
637 }
638 \cs_new_protected:Npn \__template_parse_vars_elt_auxv:nnn #1#2#3
639 {
640   \tl_if_single:nTF {#1}
641   {
642     \cs_if_exist:NF #1
643     { \use:c { \__template_map_var_type: _new:N } #1 }
644     \prop_put:NVn \l__template_vars_prop

```

```

645         \l__template_key_name_tl {#3#1}
646     }
647     { \msg_error:nnn { template } { bad-variable } {#3#1} }
648 }
649 \cs_generate_variant:Nn \__template_parse_vars_elt_auxv:nnn { e }

```

(End of definition for __template_parse_vars_elt:nnn and others.)

__template_parse_vars_choice:nnn
__template_parse_vars_function:nnn
__template_parse_vars_instance:nnn

The various special cases.

```

650 \cs_new_protected:Npn \__template_parse_vars_choice:nnn #1#2#3
651 { \__template_implement_choices:nn {#2} {#1} }
652 \cs_new_protected:Npn \__template_parse_vars_function:nnn #1#2#3
653 {
654     \cs_if_exist:NF #1
655     { \cs_new:Npn #1 { } }
656     \prop_put:NVn \l__template_vars_prop
657     \l__template_key_name_tl {#3#1}
658 }
659 \cs_new_protected:Npn \__template_parse_vars_instance:nnn #1#2#3
660 {
661     \prop_put:NVn \l__template_vars_prop
662     \l__template_key_name_tl {#3#1}
663 }

```

(End of definition for __template_parse_vars_choice:nnn, __template_parse_vars_function:nnn, and __template_parse_vars_instance:nnn.)

__template_map_var_type: Turn a “friendly” variable type into an expl3 one.

```

664 \cs_new:Npn \__template_map_var_type:
665 {
666     \str_case:Vn \l__template_keytype_tl
667     {
668         { boolean } { bool }
669         { commalist } { clist }
670         { integer } { int }
671         { length } { dim }
672         { muskip } { muskip }
673         { real } { fp }
674         { skip } { skip }
675         { tokenlist } { tl }
676     }
677 }

```

(End of definition for __template_map_var_type:.)

__template_implement_choices:nn
__template_implement_choices_default:

Implementing choices requires a second key–value loop. So after a little set-up, the standard parser is called.

```

678 \cs_new_protected:Npn \__template_implement_choices:nn #1#2
679 {
680     \clist_set:NV \l__template_tmp_clist \l__template_keytype_arg_tl
681     \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl { }
682     \keyval_parse:nnn
683     { \__template_implement_choice_elt:n }
684     { \__template_implement_choice_elt:nnn {#1} }
685     {#2}

```

```

686 \prop_get:NVNT \l__template_values_prop \l__template_key_name_tl
687 \l__template_tmp_tl
688 { \__template_implement_choices_default: }
689 \clist_if_empty:NF \l__template_tmp_clist
690 {
691   \clist_map_inline:Nn \l__template_tmp_clist
692     { \msg_error:nnn { template } { choice-not-implemented } {##1} }
693 }
694 }

```

A sanity check for the default value, so that an error is raised now and not when converting to assignments.

```

695 \cs_new_protected:Npn \__template_implement_choices_default:
696 {
697   \tl_set:Nc \l__template_tmp_tl
698     { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
699   \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
700   {
701     \tl_set:Nc \l__template_tmp_tl
702       { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
703     \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
704     {
705       \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
706         \l__template_tmp_tl
707       \__template_split_keytype_arg:V \l__template_tmp_tl
708       \prop_get:NVN \l__template_values_prop \l__template_key_name_tl
709         \l__template_tmp_tl
710       \msg_error:nnVV { template } { unknown-default-choice }
711         \l__template_key_name_tl
712         \l__template_key_name_tl
713     }
714   }
715 }

```

(End of definition for __template_implement_choices:nn and __template_implement_choices-default:.)

```

\__template_implement_choice_elt:nnn
\__template_implement_choice_elt_aux:nnn
\__template_implement_choice_elt_aux:n
\__template_implement_choice_elt:n

```

The actual storage of the implementation of a choice is mainly about error checking. The code here ensures that all choices have to have been declared, apart from the special **unknown** choice, which must come last. The code for each choice is stored along with the key name in the variables property list.

```

716 \cs_new_protected:Npn \__template_implement_choice_elt:nnn #1#2#3
717 {
718   \clist_if_empty:NTF \l__template_tmp_clist
719   {
720     \str_if_eq:nnTF {#2} { unknown }
721     { \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3} }
722     { \__template_implement_choice_elt_aux:n {#2} }
723   }
724   {
725     \clist_if_in:NnTF \l__template_tmp_clist {#2}
726     {
727       \clist_remove_all:Nn \l__template_tmp_clist {#2}
728       \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3}

```

```

729         }
730         { \__template_implement_choice_elt_aux:n {#2} }
731     }
732 }
733 \cs_new_protected:Npn \__template_implement_choice_elt_aux:n #1
734 {
735     \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
736     \l__template_tmp_tl
737     \__template_split_keytype_arg:V \l__template_tmp_tl
738     \msg_error:nnVn { template } { unknown-choice } \l__template_key_name_tl {#1}
739 }
740 \cs_new_protected:Npn \__template_implement_choice_elt_aux:nnn #1#2#3
741 {
742     \tl_set:Ne \l__template_tmp_tl
743     { \l__template_key_name_tl \c_space_tl #2 }
744     \prop_put:NVN \l__template_vars_prop \l__template_tmp_tl {#3}
745 }
746 \cs_new_protected:Npn \__template_implement_choice_elt:n #1
747 {
748     \msg_error:nnVn { template } { choice-requires-code }
749     \l__template_key_name_tl {#1}
750 }

```

(End of definition for __template_implement_choice_elt:nnn and others.)

13.10 Editing template defaults

__template_edit_defaults:nnn

Editing the template defaults means getting the values back out of the store, then parsing the list of new values before putting the updated list back into storage.

```

751 \cs_new_protected:Npn \__template_edit_defaults:nnn #1#2#3
752 {
753     \__template_if_keys_exist:nnT {#1} {#2}
754     {
755         \__template_recover_defaults:nn {#1} {#2}
756         \__template_parse_values:nnn {#1} {#2} {#3}
757         \__template_store_defaults:nn {#1} {#2}
758     }
759 }

```

(End of definition for __template_edit_defaults:nnn.)

__template_parse_values:nnn

__template_parse_known_values:nnn

__template_parse_values:NNnnn

The routine to parse values is the same for both editing a template and setting up an instance. So the code here does only the minimum necessary for reading the values. To allow code sharing, there are two code paths for a missing key: error here or store the information for later processing.

```

760 \cs_new_protected:Npn \__template_parse_values:nnn
761 { \__template_parse_values:NNnnn \use_i:nn \__template_parse_unknown_error:Vn }
762 \cs_new_protected:Npn \__template_parse_known_values:nnn
763 { \__template_parse_values:NNnnn \use_ii:nn \__template_parse_unknown_store:Vn }
764 \cs_new_protected:Npn \__template_parse_values:NNnnn #1#2#3#4#5
765 {
766     \__template_recover_keytypes:nn {#3} {#4}
767     \seq_clear:N \l__template_value_order_seq
768     \keyval_parse:nnn

```

```

769     { \__template_parse_values_elt:Nnn #1#2 }
770     { \__template_parse_values_elt:Nnn #2 }
771     {#5}
772 }

```

(End of definition for __template_parse_values:nnn, __template_parse_known_values:nnn, and __template_parse_values:NNnnn.)

__template_parse_values_elt:Nnn Almost key needs a value, so this is just an error routine other than for booleans.

```

773 \cs_new_protected:Npn \__template_parse_values_elt:Nnn #1#2#3
774 {
775   \prop_get:NnN \l__template_keytypes_prop {#3} \l__template_tmp_tl
776   \str_if_eq:VnTF \l__template_tmp_tl { boolean }
777   { \__template_parse_values_elt:Nnn #2 {#3} { true } }
778   {
779     #1
780     {
781       \bool_set_true:N \l__template_error_bool
782       \msg_error:nnn { template } { key-no-value } {#3}
783     }
784     { \clist_put_right:Nn \UnusedTemplateKeys {#3} }
785   }
786 }

```

(End of definition for __template_parse_values_elt:Nnn.)

__template_parse_values_elt:Nnn To store the value, find the keytype then call the saving function. These need the current key name, stored in \l__template_key_name_tl.

```

787 \cs_new_protected:Npn \__template_parse_values_elt:Nnn #1#2#3
788 {
789   \use:e
790   {
791     \__template_parse_values_elt_auxi:w
792     \tl_trim_spaces:e { \tl_to_str:n { #2 : n : } }
793     \exp_not:N \q_stop
794   }
795   \cs_if_exist:cTF { __template_parse_values_set: \l__template_value_exp_str }
796   {
797     \prop_get:NvNTF \l__template_keytypes_prop \l__template_key_name_tl
798     \l__template_tmp_tl
799     { \__template_parse_values_elt_auxii:nn { set } }
800     { \__template_parse_values_elt_auxiii:Nn #1 }
801     {#3}
802   }
803   {
804     \msg_error:nnV { template } { unknown-expansion }
805     \l__template_value_exp_str
806   }
807 }
808 \use:e
809 {
810   \cs_new_protected:Npn \exp_not:N \__template_parse_values_elt_auxi:w
811   #1 \token_to_str:N : #2 \token_to_str:N : #3 \exp_not:N \q_stop
812 }
813 {

```

```

814     \tl_set:Nn \l__template_key_name_tl {#1}
815     \str_set:Nn \l__template_value_exp_str {#2}
816   }
817 \cs_new_protected:Npn \__template_parse_values_elt_auxii:nn #1#2
818 {
819   \__template_split_keytype_arg:V \l__template_tmp_tl
820   \cs_if_exist:cTF
821     { __template_parse_values_ \l__template_keytype_tl :nn }
822     {
823       \prop_get:NVNT \l__template_msg_prop \l__template_key_name_tl
824       \l__template_tmp_tl
825       { \l__template_tmp_tl }
826       \use:c { __template_parse_values_ \l__template_keytype_tl :nn } {#1}
827     }
828     { \use:c { __template_parse_values_ #1 : \l__template_value_exp_str } }
829     {#2}
830 }

```

Hunt for a trailing +: this allows the += syntax, treating the token as part of the mid-part not the key name.

```

831 \cs_new_protected:Npn \__template_parse_values_elt_auxiii:Nn #1#2
832 {
833   \exp_after:wN \__template_parse_values_elt_auxiv:w
834   \l__template_key_name_tl \q_nil + \q_nil \q_stop {#2} #1
835 }
836 \cs_new_protected:Npn \__template_parse_values_elt_auxiv:w #1 + \q_nil #2 \q_stop #3#4
837 {
838   \tl_if_empty:nTF {#2}
839     { #4 \l__template_key_name_tl }
840     {
841       \tl_set:Ne \l__template_key_name_tl { \tl_trim_spaces:n {#1} }
842       \prop_get:NVNTF \l__template_keytypes_prop \l__template_key_name_tl
843       \l__template_tmp_tl
844       { \__template_parse_values_elt_auxii:nn { add } }
845       { #4 \l__template_key_name_tl }
846     }
847     {#3}
848 }

```

Equivalent keys need to be “recycled” back around.

```

849 \cs_new_protected:Npn \__template_parse_values_alias:nn #1#2
850 {
851   \seq_remove_all:NV \l__template_value_order_seq \l__template_key_name_tl
852   \prop_get:NVN \l__template_vars_prop \l__template_key_name_tl
853   \l__template_tmp_tl
854   \tl_set_eq:NN \l__template_key_name_tl \l__template_tmp_tl
855   \__template_parse_values_alias_aux:nn {#2} {#1}
856 }
857 \cs_new_protected:Npn \__template_parse_values_alias_aux:nn #1#2
858 {
859   \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
860   \l__template_keytype_tl
861   \__template_parse_values_elt_auxii:nn {#2} {#1}
862 }
863 \cs_new_protected:Npn \__template_parse_values_meta:nn #1#2

```

```

864 {
865   \seq_remove_all:NV \l__template_value_order_seq \l__template_key_name_tl
866   \prop_get:NVN \l__template_vars_prop \l__template_key_name_tl
867   \l__template_tmp_tl
868   \keyval_parse:nnV
869   { \msg_error:nnn { template } { key-no-value } }
870   { \__template_parse_values_meta_elt:nnnn {#1} {#2} }
871   \l__template_tmp_tl
872 }
873 \cs_new_protected:Npn \__template_parse_values_meta_elt:nnnn #1#2#3#4
874 {
875   \tl_set:Nn \l__template_key_name_tl {#3}
876   \cs_set:Npn \__template_tmp:n ##1 {#4}
877   \tl_set:Ne \l__template_tmp_tl { \__template_tmp:n {#2} }
878   \exp_args:NV \__template_parse_values_alias_aux:nn
879   \l__template_tmp_tl {#1}
880 }

```

Setting and adding start quite similar, but diverge later.

```

881 \cs_new_protected:Npn \__template_parse_values_set:n #1
882 {
883   \seq_put_right:NV \l__template_value_order_seq \l__template_key_name_tl
884   \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1}
885 }
886 \cs_generate_variant:Nn \__template_parse_values_set:n { o , V , v , e }
887 \cs_new_eq:NN \__template_parse_values_set:N \__template_parse_values_set:n
888 \cs_generate_variant:Nn \__template_parse_values_set:N { c }
889 \cs_new_protected:Npn \__template_parse_values_add:n #1
890 {
891   \seq_put_right:NV \l__template_value_order_seq \l__template_key_name_tl
892   \prop_get:NVNF \l__template_values_prop \l__template_key_name_tl
893   \l__template_tmp_tl
894   { \tl_clear:N \l__template_tmp_tl }
895   \use:c { __template_parse_values_add_ \l__template_keytype_tl :n } {#1}
896 }
897 \cs_generate_variant:Nn \__template_parse_values_add:n { o , V , v , e }
898 \cs_new_eq:NN \__template_parse_values_add:N \__template_parse_values_add:n
899 \cs_generate_variant:Nn \__template_parse_values_add:N { c }
900 \clist_map_inline:nn
901 { boolean , choice , function , instance }
902 {
903   \cs_new_protected:cpn { __template_parse_values_add_ #1 :n } ##1
904   { \msg_error:nnn { template } { augmented-key-inappropriate } {#1} }
905 }
906 \clist_map_variable:nNn
907 { integer , length , muskip , real , skip }
908 \l__template_keytype_tl
909 {
910   \cs_new_protected:cpe
911   { __template_parse_values_add_ \l__template_keytype_tl :n } #1
912   {
913     \tl_set:Ne \exp_not:N \l__template_tmp_tl
914     {
915       \exp_not:c { __template_map_var_type: _eval:n }
916       { \exp_not:N \l__template_tmp_tl + #1 }
917     }
918   }
919 }

```

```

917     }
918     \prop_put:NVV \exp_not:N \l__template_values_prop
919     \exp_not:N \l__template_key_name_tl \exp_not:N \l__template_tmp_tl
920   }
921 }
922 \cs_new_protected:Npn \__template_parse_values_add_commalist:n #1
923 {
924   \clist_put_right:Nn \l__template_tmp_tl {#1}
925   \prop_put:NVV \l__template_values_prop \l__template_key_name_tl \l__template_tmp_tl
926 }
927 \cs_new_protected:Npn \__template_parse_values_add_tokenlist:n #1
928 {
929   \tl_put_right:Nn \l__template_tmp_tl {#1}
930   \prop_put:NVV \l__template_values_prop \l__template_key_name_tl \l__template_tmp_tl
931 }
932 \cs_new_protected:Npn \__template_parse_unknown_error:nn #1#2
933 { \msg_error:nnn { template } { unknown-key } {#1} }
934 \cs_generate_variant:Nn \__template_parse_unknown_error:nn { V }
935 \cs_new_protected:Npn \__template_parse_unknown_store:nn #1#2
936 { \clist_put_right:Nn \UnusedTemplateKeys { #1 = {#2} } }
937 \cs_generate_variant:Nn \__template_parse_unknown_store:nn { V }

```

(End of definition for `\__template_parse_values_elt:Nnn` and others.)

`\__template_template_set_eq:nnn`

To copy a template, each of the lists plus the code has to be copied across. To keep this independent of the list storage system, it is all done with two-part shuffles.

```

938 \cs_new_protected:Npn \__template_template_set_eq:nnn #1#2#3
939 {
940   \__template_recover_defaults:nn {#1} {#3}
941   \__template_store_defaults:nn {#1} {#2}
942   \__template_recover_keytypes:nn {#1} {#3}
943   \__template_store_keytypes:nn {#1} {#2}
944   \__template_recover_vars:nn {#1} {#3}
945   \__template_store_vars:nn {#1} {#2}
946   \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
947   { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
948   \cs_gset_eq:cc { \c__template_code_root_tl #1 / #2 }
949   { \c__template_code_root_tl #1 / #3 }
950 }

```

(End of definition for `\__template_template_set_eq:nnn`.)

13.11 Creating instances of templates

`\__template_declare_instance:nnnn`
`\__template_declare_instance_aux:nnnn`

Making an instance has two distinct parts. First, the keys given are parsed to transfer the values into the structured data format used internally. This allows the default and given values to be combined with no repetition. In the second step, the structured data is converted to pre-defined variable assignments, and these are stored in the function for the instance.

```

951 \cs_new_protected:Npn \__template_declare_instance:nnnn #1#2#3#4
952 {
953   \__template_execute_if_code_exist:nnT {#1} {#2}
954   {
955     \__template_recover_defaults:nn {#1} {#2}

```

```

956     \__template_recover_vars:nn {#1} {#2}
957     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
958   }
959 }
960 \cs_new_protected:Npn \__template_declare_instance_aux:nnnn #1#2#3#4
961 {
962   \bool_set_false:N \l__template_error_bool
963   \__template_parse_values:nnn {#1} {#2} {#4}
964   \bool_if:NF \l__template_error_bool
965   {
966     \prop_put:Nnn \l__template_values_prop { from~template } {#2}
967     \__template_store_values:nn {#1} {#3}
968     \__template_convert_to_assignments:N \l__template_key_order_seq
969     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #3 }
970     { \msg_info:nnnn { template } { declare-instance } {#3} {#1} }
971     \cs_set_protected:cpe { \c__template_instances_root_tl #1 / #3 }
972     {
973       \tl_set:Nn \exp_not:N \l__template_current_type_tl {#1}
974       \tl_set:Nn \exp_not:N \l__template_current_template_tl {#2}
975       \exp_not:N \__template_assignments_push:n
976       { \exp_not:N \l__template_assignments_tl }
977       \exp_not:c { \c__template_code_root_tl #1 / #2 }
978     }
979   }
980 }

```

(End of definition for __template_declare_instance:nnnn and __template_declare_instance_aux:nnnn.)

__template_instance_set_eq:nnn Copy-paste an instance.

```

981 \cs_new_protected:Npn \__template_instance_set_eq:nnn #1#2#3
982 {
983   \__template_if_instance_exist:nnTF {#1} {#3}
984   {
985     \__template_recover_values:nn {#1} {#3}
986     \__template_store_values:nn {#1} {#2}
987     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #2 }
988     { \msg_info:nnnn { template } { declare-instance } {#2} {#1} }
989     \cs_set_eq:cc { \c__template_instances_root_tl #1 / #2 }
990     { \c__template_instances_root_tl #1 / #3 }
991   }
992   { \msg_error:nnnn { template } { unknown-instance } {#1} {#3} }
993 }

```

(End of definition for __template_instance_set_eq:nnn.)

__template_edit_instance:nnn Editing an instance is almost identical to declaring one. The only variation is the source of the values to use. When editing, they are recovered from the previous instance run.

```

\__template_edit_instance_aux:nnnn
\__template_edit_instance_aux:nVnnn
994 \cs_new_protected:Npn \__template_edit_instance:nnn #1#2#3
995 {
996   \__template_if_instance_exist:nnTF {#1} {#2}
997   {
998     \__template_recover_values:nn {#1} {#2}
999     \prop_get:NnN \l__template_values_prop { from~template }
1000     \l__template_tmp_tl

```

```

1001         \__template_edit_instance_aux:nVnn
1002         {#1} \l__template_tmp_tl {#2} {#3}
1003     }
1004     { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
1005 }
1006 \cs_new_protected:Npn \__template_edit_instance_aux:nnnn #1#2#3#4
1007 {
1008     \__template_recover_vars:nn {#1} {#2}
1009     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
1010 }
1011 \cs_generate_variant:Nn \__template_edit_instance_aux:nnnn { nV }

```

(End of definition for __template_edit_instance:nnn and __template_edit_instance_aux:nnnn.)

The idea on converting to a set of assignments is to loop over each key, so that the loop order follows the declaration order of the keys. This is done using a sequence as property lists are not “ordered”.

```

1012 \cs_new_protected:Npn \__template_convert_to_assignments:N #1
1013 {
1014     \tl_clear:N \l__template_assignments_tl
1015     \seq_map_function:NN #1
1016     \__template_convert_to_assignments_aux:n
1017 }
1018 \cs_new_protected:Npn \__template_convert_to_assignments_aux:n #1
1019 {
1020     \prop_get:NnN \l__template_keytypes_prop {#1} \l__template_tmp_tl
1021     \__template_convert_to_assignments_aux:nV {#1} \l__template_tmp_tl
1022 }

```

The second auxiliary function actually does the work. The arguments here are the key name (#1) and the keytype (#2). From those, the value to assign and the name of the appropriate variable are recovered. A bit of work is then needed to sort out keytypes with arguments (for example instances), and to look for global assignments. Once that is done, a hand-off can be made to the handler for the relevant keytype.

```

1023 \cs_new_protected:Npn \__template_convert_to_assignments_aux:nn #1#2
1024 {
1025     \prop_pop:NnNT \l__template_values_prop {#1} \l__template_value_tl
1026     {
1027         \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_var_tl
1028         {
1029             \tl_set:Nn \l__template_key_name_tl {#1}
1030             \__template_split_keytype_arg:n {#2}
1031             \bool_lazy_any:nF
1032             {
1033                 { \str_if_eq_p:Vn \l__template_keytype_tl { alias } }
1034                 { \str_if_eq_p:Vn \l__template_keytype_tl { choice } }
1035                 { \str_if_eq_p:Vn \l__template_keytype_tl { meta } }
1036             }
1037             { \__template_find_global: }
1038             \cs_if_exist_use:cF { __template_assign_ \l__template_keytype_tl : }
1039             { \__template_assign_variable: }
1040         }
1041         { \msg_error:nnn { template } { unknown-key } {#1} }
1042     }
}

```

```

1043 }
1044 \cs_generate_variant:Nn \__template_convert_to_assignments_aux:nn { nV }

```

(End of definition for __template_convert_to_assignments:N, __template_convert_to_assignments_aux:n, and __template_convert_to_assignments_aux:nn.)

__template_find_global: Global assignments should have the phrase `global` at the front. This is pretty easy to find: no other error checking, though.

```

1045 \cs_new_protected:Npn \__template_find_global:
1046 {
1047   \bool_set_false:N \l__template_global_bool
1048   \tl_if_in:ont \l__template_var_tl { global }
1049   {
1050     \exp_after:wN \__template_find_global_aux:w \l__template_var_tl \s__template_stop
1051   }
1052 }
1053 \cs_new_protected:Npn \__template_find_global_aux:w #1 global #2 \s__template_stop
1054 {
1055   \tl_set:Nn \l__template_var_tl {#2}
1056   \bool_set_true:N \l__template_global_bool
1057 }

```

(End of definition for __template_find_global: and __template_find_global_aux:w.)

__template_instance_value:nnn For editing templates.

```

1058 \cs_new:Npn \__template_instance_value:nnn #1#2#3
1059 {
1060   \__template_if_instance_exist:nnT {#1} {#2}
1061   { \prop_item:cn { \c__template_values_root_tl #1 / #2 } {#3} }
1062 }

```

(End of definition for __template_instance_value:nnn.)

13.12 Using templates directly

__template_use_template:nnn Directly use a template with a particular parameter setting. This is also picked up if used in a nested fashion inside a parameter list. The idea is essentially the same as creating an instance, just with no saving of the result.

```

1063 \cs_new_protected:Npn \__template_use_template:nnn #1#2#3
1064 {
1065   \__template_execute_if_code_exist:nnT {#1} {#2}
1066   {
1067     \__template_recover_defaults:nn {#1} {#2}
1068     \__template_recover_vars:nn {#1} {#2}
1069     \__template_parse_values:nnn {#1} {#2} {#3}
1070     \__template_convert_to_assignments:N \l__template_key_order_seq
1071     \__template_debug_typeout:n
1072     { Using~template~'#2'~of~type~'#1'~directly~ \msg_line_context: }
1073     \tl_set:Nn \l__template_current_type_tl {#1}
1074     \tl_set:Nn \l__template_current_template_tl {#2}
1075     \use:c { \c__template_code_root_tl #1 / #2 }
1076   }
1077 }

```

(End of definition for __template_use_template:nnn.)

13.13 Assigning values to variables

`__template_assign_boolean:` Setting a Boolean value is slightly different to everything else as the value can be used to work out which `set` function to call. As long as there is no need to recover things from another variable, everything is pretty easy. If there is, then we need to allow for the fact that the recovered value here will *not* be expandable, so needs to be converted to something that is.

```

1078 \cs_new_protected:Npn __template_assign_boolean:
1079 {
1080   \bool_if:NTF \l__template_global_bool
1081     { __template_assign_boolean_aux:n { bool_gset } }
1082     { __template_assign_boolean_aux:n { bool_set } }
1083 }
1084 \cs_new_protected:Npn __template_assign_boolean_aux:n #1
1085 {
1086   \__template_if_key_value:VTF \l__template_value_tl
1087   {
1088     \__template_key_to_value:
1089     \tl_put_right:Ne \l__template_assignments_tl
1090     {
1091       \exp_not:c { #1 _eq:NN }
1092       \exp_not:V \l__template_var_tl
1093       \exp_not:V \l__template_value_tl
1094     }
1095   }
1096   {
1097     \tl_put_right:Ne \l__template_assignments_tl
1098     {
1099       \exp_not:c { #1 _ \l__template_value_tl :N }
1100       \exp_not:V \l__template_var_tl
1101     }
1102   }
1103 }

```

(End of definition for `__template_assign_boolean:` and `__template_assign_boolean_aux:n`.)

`__template_assign_choice:` The idea here is to find either the choice as-given or else the special `unknown` choice, and to copy the appropriate code across.

```

__template_assign_choice_aux:nF
__template_assign_choice_aux:eF
1104 \cs_new_protected:Npn __template_assign_choice:
1105 {
1106   \__template_assign_choice_aux:eF
1107   { \l__template_key_name_tl \c_space_tl \l__template_value_tl }
1108   {
1109     \__template_assign_choice_aux:eF
1110     { \l__template_key_name_tl \c_space_tl unknown }
1111     {
1112       \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
1113       \l__template_tmp_tl
1114       \__template_split_keytype_arg:V \l__template_tmp_tl
1115       \msg_error:nnVV { template } { unknown-choice }
1116       \l__template_key_name_tl
1117       \l__template_value_tl
1118     }
1119   }

```

```

1120 }
1121 \cs_new_protected:Npn \__template_assign_choice_aux:nF #1
1122 {
1123   \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_tmp_tl
1124   { \tl_put_right:NV \l__template_assignments_tl \l__template_tmp_tl }
1125 }
1126 \cs_generate_variant:Nn \__template_assign_choice_aux:nF { e }

```

(End of definition for __template_assign_choice: and __template_assign_choice_aux:nF.)

__template_assign_function: This looks a bit messy but is only actually one function.

```

\__template_assign_function_aux:N
1127 \cs_new_protected:Npn \__template_assign_function:
1128 {
1129   \bool_if:NTF \l__template_global_bool
1130   { \__template_assign_function_aux:N \cs_gset_protected:Npn }
1131   { \__template_assign_function_aux:N \cs_set_protected:Npn }
1132 }
1133 \cs_new_protected:Npn \__template_assign_function_aux:N #1
1134 {
1135   \tl_put_right:Ne \l__template_assignments_tl
1136   {
1137     \cs_generate_from_arg_count:NNnn
1138     \exp_not:V \l__template_var_tl
1139     \exp_not:N #1
1140     { \exp_not:V \l__template_keytype_arg_tl }
1141     { \exp_not:V \l__template_value_tl }
1142   }
1143 }

```

(End of definition for __template_assign_function: and __template_assign_function_aux:N.)

__template_assign_instance: Using an instance means adding the appropriate function creation to the tl. No checks are made at this stage, so if the instance is not valid then errors will arise later.

```

\__template_assign_instance_aux:N
1144 \cs_new_protected:Npn \__template_assign_instance:
1145 {
1146   \bool_if:NTF \l__template_global_bool
1147   { \__template_assign_instance_aux:N \cs_gset_protected:Npn }
1148   { \__template_assign_instance_aux:N \cs_set_protected:Npn }
1149 }
1150 \cs_new_protected:Npn \__template_assign_instance_aux:N #1
1151 {
1152   \tl_put_right:Ne \l__template_assignments_tl
1153   {
1154     \exp_not:N #1 \exp_not:V \l__template_var_tl
1155     {
1156       \__template_use_instance:nn
1157       { \exp_not:V \l__template_keytype_arg_tl }
1158       { \exp_not:V \l__template_value_tl }
1159     }
1160   }
1161 }

```

(End of definition for __template_assign_instance: and __template_assign_instance_aux:N.)

`\__template_assign_variable:` A general-purpose function for all of the other assignments. As long as the value is not coming from another variable, the stored value is simply transferred for output. We use V-type expansion for the `\KeyValue` case: for token lists this is essential, whilst for register-based variables, it does no harm and avoids needing a low-level test.

```

1162 \cs_new_protected:Npn \__template_assign_variable:
1163 {
1164   \exp_args:Ne \__template_assign_variable:n
1165   {
1166     \__template_map_var_type:
1167     -
1168     \bool_if:NT \l__template_global_bool { g }
1169     set:N
1170   }
1171 }

```

Notice we need a V-type variant for each (g)set operation here: these need to be provided by `expl3`.

```

1172 \cs_new_protected:Npn \__template_assign_variable:n #1
1173 {
1174   \__template_if_key_value:VTF \l__template_value_tl
1175   {
1176     \__template_key_to_value:
1177     \tl_put_right:Ne \l__template_assignments_tl
1178     {
1179       \exp_not:c { #1 V } \exp_not:V \l__template_var_tl
1180       \exp_not:V \l__template_value_tl
1181     }
1182   }
1183   {
1184     \tl_put_right:Ne \l__template_assignments_tl
1185     {
1186       \exp_not:c { #1 n } \exp_not:V \l__template_var_tl
1187       { \exp_not:V \l__template_value_tl }
1188     }
1189   }
1190 }

```

(End of definition for `\__template_assign_variable:` and `\__template_assign_variable:n`.)

`\__template_key_to_value:` The idea here is to recover the attribute value of another key. To do that, the marker is removed and a look up takes place. If this is successful, then the name of the variable of the attribute is returned. This assumes that the value will be used in context where it will be converted to a value, for example when setting a number. There is also a need to check in case the copied value happens to be global.

```

1191 \cs_new_protected:Npn \__template_key_to_value:
1192 { \exp_after:wN \__template_key_to_value_auxi:w \l__template_value_tl }
1193 \cs_new_protected:Npn \__template_key_to_value_auxi:w \KeyValue #1
1194 {
1195   \tl_set:Ne \l__template_tmp_tl { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
1196   \prop_get:NVNTF \l__template_vars_prop \l__template_tmp_tl
1197   \l__template_value_tl
1198   {
1199     \exp_after:wN \__template_key_to_value_auxii:w \l__template_value_tl
1200     \s__template_mark global \q__template_nil \s__template_stop

```

```

1201     }
1202     { \msg_error:nnV { template } { unknown-attribute } \l__template_tmp_tl }
1203   }
1204   \cs_new_protected:Npn \__template_key_to_value_auxii:w #1 global #2#3 \s__template_stop
1205   {
1206     \__template_quark_if_nil:NF #2
1207     { \tl_set:Nn \l__template_value_tl {#2} }
1208   }

```

(End of definition for __template_key_to_value:, __template_key_to_value_auxi:w, and __template_key_to_value_auxii:w.)

13.14 Using instances

__template_use_instance:nn Using an instance is just a question of finding the appropriate function. If nothing is found, an error is raised. One complication is that if the first token of argument #2 is \UseTemplate then that is also valid. There is an error-test to make sure that the types agree, and if so the template is used directly.

```

1209 \cs_new_protected:Npn \__template_use_instance:nn #1#2
1210 {
1211   \__template_if_use_template:nTF {#2}
1212   { \__template_use_instance_aux:nNnnn {#1} #2 }
1213   { \__template_use_instance_auxi:nn {#1} {#2} }
1214 }
1215 \cs_new_protected:Npn \__template_use_instance_aux:nNnnn #1#2#3#4#5
1216 {
1217   \str_if_eq:nnTF {#1} {#3}
1218   { \__template_use_template:nnn {#3} {#4} {#5} }
1219   { \msg_error:nnnn { template } { type-mismatch } {#1} {#3} }
1220 }
1221 \cs_new_protected:Npn \__template_use_instance_auxi:nn #1#2
1222 {
1223   \__template_if_instance_exist:nnTF {#1} {#2}
1224   { \__template_use_instance_auxii:nn {#1} {#2} }
1225   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
1226 }
1227 \cs_new_protected:Npn \__template_use_instance_auxii:nn #1#2
1228 {
1229   \__template_debug_typeout:n
1230   { Using~instance~'#2'~of~type~'#1'~ \msg_line_context: }
1231   \use:c { \c__template_instances_root_tl #1 / #2 }
1232 }

```

(End of definition for __template_use_instance:nn and others.)

13.15 Assignment manipulation

A few functions to transfer assignments about, as this is needed by \AssignTemplateKeys.

__template_assignments_pop: To actually use the assignments.

```

1233 \cs_new:Npn \__template_assignments_pop: { \l__template_assignments_tl }

```

(End of definition for __template_assignments_pop:.)

```

\__template_assignments_push:n Here, the assignments are stored for later use.
1234 \cs_new_protected:Npn \__template_assignments_push:n #1
1235 { \tl_set:Nn \l__template_assignments_tl {#1} }

```

(End of definition for __template_assignments_push:n.)

13.16 Ad hoc key setting

For (mainly) user-overrides at point of use. We aim to keep the key structures “inside” ltemplates.

__template_keys_set:nnn Setting keys in the document body requires that we work in a similar way to creating instances. To have keys set in source order, we use a sequence which is set during parsing. This allows us to reuse internals. Note that the key argument is expanded exactly once, which allows for users “recycling” keys as \UnusedTemplateKeys directly rather than by value.

```

1236 \cs_new_protected:Npn \__template_keys_set:nnn #1#2#3
1237 {
1238   \__template_keys_set_known:nnn {#1} {#2} {#3}
1239   \tl_if_empty:NF \UnusedTemplateKeys
1240   {
1241     \msg_error:nnV { template } { unknown-ad-hoc-keys }
1242     \UnusedTemplateKeys
1243   }
1244 }
1245 \cs_new_protected:Npn \__template_keys_set_known:nnn #1#2#3
1246 { \exp_args:No \__template_keys_set_known_aux:nnn {#3} {#1} {#2} }

```

Key setting is skipped if the input is entirely blank or if it is exactly \NoValue (which could happen if material is passed at a higher level without a branching check). As well as setting native keys, we also check for any in l3keys for compatibility with the previous approach taken.

```

1247 \cs_new_protected:Npn \__template_keys_set_known_aux:nnn #1#2#3
1248 {
1249   \tl_clear:N \UnusedTemplateKeys
1250   \bool_lazy_or:nnF
1251   { \tl_if_blank_p:n {#1} }
1252   { \tl_if_novalue_p:n {#1} }
1253   {
1254     \keys_set_known:nnN { template / #2 / #3 } {#1} \l__template_tmp_tl
1255     \__template_recover_vars:nn {#2} {#3}
1256     \exp_args:NnnV \__template_parse_known_values:nnn {#2} {#3} \l__template_tmp_tl
1257     \__template_convert_to_assignments:N \l__template_value_order_seq
1258     \tl_use:N \l__template_assignments_tl
1259   }
1260 }

```

(End of definition for __template_keys_set:nnn, __template_keys_set_known:nnn, and __template_keys_set_known_aux:nnn.)

13.17 Showing templates and instances

`\__template_show_code:nn` Showing the code for a template is just a translation of `\cs_show:c`.

```
1261 \cs_new_protected:Npn \__template_show_code:nn #1#2
1262 { \cs_show:c { \c__template_code_root_tl #1 / #2 } }
```

(End of definition for __template_show_code:nn.)

`\__template_show_defaults:nn` A modified version of the property-list printing code, such that the output refers to templates and instances rather than to the underlying structures.

```
\__template_show_keytypes:nn
\__template_show_vars:nn
\__template_show:Nnnn
1263 \cs_new_protected:Npn \__template_show_defaults:nn #1#2
1264 {
1265   \__template_if_keys_exist:nnT {#1} {#2}
1266   {
1267     \__template_recover_defaults:nn {#1} {#2}
1268     \__template_show:Nnnn \l__template_values_prop
1269     {#1} {#2} { default~values }
1270   }
1271 }
1272 \cs_new_protected:Npn \__template_show_keytypes:nn #1#2
1273 {
1274   \__template_if_keys_exist:nnT {#1} {#2}
1275   {
1276     \__template_recover_keytypes:nn {#1} {#2}
1277     \__template_show:Nnnn \l__template_keytypes_prop
1278     {#1} {#2} { interface }
1279   }
1280 }
1281 \cs_new_protected:Npn \__template_show_vars:nn #1#2
1282 {
1283   \__template_execute_if_code_exist:nnT {#1} {#2}
1284   {
1285     \__template_recover_vars:nn {#1} {#2}
1286     \__template_show:Nnnn \l__template_vars_prop
1287     {#1} {#2} { variable~mapping }
1288   }
1289 }
1290 \cs_new_protected:Npn \__template_show:Nnnn #1#2#3#4
1291 {
1292   \msg_show:nneeee { template } { show-attribute }
1293   { \tl_to_str:n {#2} }
1294   { \tl_to_str:n {#3} }
1295   { \tl_to_str:n {#4} }
1296   { \prop_map_function:NN #1 \msg_show_item_unbraced:nn }
1297 }
```

(End of definition for __template_show_defaults:nn and others.)

`\__template_show_values:nn` Instance values are a little more complex, as is the template to consider.

```
1298 \cs_new_protected:Npn \__template_show_values:nn #1#2
1299 {
1300   \__template_if_instance_exist:nnTF {#1} {#2}
1301   {
1302     \__template_recover_values:nn {#1} {#2}
```

```

1303     \msg_show:nneee { template } { show-values }
1304     { \tl_to_str:n {#1} }
1305     { \tl_to_str:n {#2} }
1306     {
1307         \prop_map_function:NN \l__template_values_prop
1308         \msg_show_item_unbraced:nn
1309     }
1310 }
1311 { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
1312 }

```

(End of definition for __template_show_values:nn.)

13.18 Messages

The text for error messages: short and long text for all of them.

```

1313 \msg_new:nnnn { template } { argument-number-mismatch }
1314 { Template~type~'~#1'~takes~#2~argument(s). }
1315 {
1316     Templates-of~type~'~#1'~require~#2~argument(s).\
1317     You-have~tried~to~make~a~template~for~'~#1'~
1318     with~#3~argument(s),~which~is~not~possible:~
1319     the~number~of~arguments~must~agree.
1320 }

1321 \msg_new:nnnn { template } { augmented-key-inappropriate }
1322 { Augmented-key~syntax~"<key>~+=~<value>"~inappropriate. }
1323 {
1324     Augmented-key~settings~does~not~apply~to~keys~of~the~type~'~#1'~.
1325 }

1326 \msg_new:nnnn { template } { bad-number-of-arguments }
1327 { Bad~number~of~arguments~for~template~type~'~#1'~. }
1328 {
1329     A~template~may~accept~between~0~and~9~arguments.\
1330     You~asked~to~use~#2~arguments:~this~is~not~supported.
1331 }

1332 \msg_new:nnnn { template } { bad-variable }
1333 { Incorrect~variable~description~'~#1'~. }
1334 {
1335     The~argument~'~#1'~is~not~of~the~form \
1336     ~~~<variable>' \
1337     ~or~ \
1338     ~~~global~<variable>' \
1339     It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
1340 }

1341 \msg_new:nnnn { template } { choice-not-implemented }
1342 { The~choice~'~#1'~has~no~implementation. }
1343 {
1344     Each~choice~listed~in~the~interface~for~a~template~must~
1345     have~an~implementation.
1346 }

1347 \msg_new:nnnn { template } { choice-no-code }
1348 { The~choice~'~#1'~requires~implementation~details. }
1349 {

```

```

1350     When~creating~template~code~using~\DeclareTemplateCode,~
1351     each~choice~name~must~have~an~associated~implementation.\\
1352     This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1353 }
1354 \msg_new:nnnn { template } { choice-requires-code }
1355 { The~choice~'#2'~for~key~'#1'~requires~an~implementation. }
1356 {
1357     You~should~have~put:\\
1358     \ \ #1::~choice~{~#2 = <code> ~} \\
1359     but~LaTeX~did~not~find~any~<code>.
1360 }

1361 \msg_new:nnnn { template } { deprecated-equiv }
1362 { The~key~equivalent~"#3"~ #2 ~deprecated~on~#1~\msg_line_context:. }
1363 {
1364     The~key~equivalent~"#3"~ #2 ~deprecated,~and~since~#1~should~
1365     not~be~used:~you~will~need~to~update~your~source.
1366 }
1367 \msg_new:nnnn { template } { duplicate-key-interface }
1368 { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
1369 {
1370     Each~key~can~only~have~one~interface~declared~in~a~template.\\
1371     LaTeX~found~two~interfaces~for~'#1'.
1372 }

1373 \msg_new:nnnn { template } { equiv-key-interface }
1374 {
1375     Key~'#1'~appears~in~interface~definition~and~this~name~therefore~
1376     cannot~be~used~as~an~equivalent~\msg_line_context:.
1377 }
1378 {
1379     Each~key~can~only~have~one~interface~declared~in~a~template.\\
1380     LaTeX~found~two~interfaces~for~'#1':~a~base~one~and~an~equivalent.
1381 }
1382 \msg_new:nnnn { template } { keytype-requires-argument }
1383 { The~key~type~'#1'~requires~an~argument~\msg_line_context:. }
1384 {
1385     You~should~have~put:\\
1386     \ \ <key-name>::~#1~{~<argument>~} \\
1387     but~LaTeX~did~not~find~an~<argument>.
1388 }

1389 \msg_new:nnnn { template } { key-no-equiv }
1390 { The~key~'#1'~has~no~equivalents~\msg_line_context:. }
1391 {
1392     When~creating~a~compatibility~key~for~a~template~
1393     every~key~listed~must~include~an~equivalent:\\
1394     \ \ <old-key>::~<type>::~~<new-key(s)>
1395 }
1396 \msg_new:nnnn { template } { key-no-value }
1397 { The~key~'#1'~has~no~value~\msg_line_context:. }
1398 {
1399     When~creating~an~instance~of~a~template~
1400     every~key~listed~must~include~a~value:\\
1401     \ \ <key>::~~<value>

```

```

1402 }
1403 \msg_new:nnnn { template } { key-no-variable }
1404 { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
1405 {
1406   When~creating~template~code~using~\DeclareTemplateCode,~
1407   each~key~name~must~have~an~associated~implementation.\\
1408   This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1409 }
1410 \msg_new:nnnn { template } { key-not-implemented }
1411 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1412 {
1413   The~definition~of~key~implementations~for~template~'#2'~
1414   of~template~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1415   The~key~was~declared~in~the~interface~definition,~
1416   and~so~an~implementation~is~required.
1417 }
1418 \msg_new:nnnn { template } { missing-keytype }
1419 { The~key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1420 {
1421   Key~interface~definitions~should~be~of~the~form\\
1422   \ \ #1~::~<key-type>\\
1423   but~LaTeX~could~not~find~a~<key-type>.
1424 }
1425 \msg_new:nnnn { template } { no-template-code }
1426 {
1427   The~template~'#2'~of~type~'#1'~is~unknown~
1428   or~has~no~implementation.
1429 }
1430 {
1431   There~is~no~code~available~for~the~template~name~given.\\
1432   This~should~be~given~using~\DeclareTemplateCode.
1433 }
1434 \msg_new:nnnn { template } { type-already-defined }
1435 { Template~type~'#1'~already~defined. }
1436 {
1437   You~have~used~\NewTemplateType~
1438   with~a~template~type~that~has~already~been~defined.
1439 }
1440 \msg_new:nnnn { template } { type-mismatch }
1441 { Template~types~'#1'~and~'#2'~do~not~agree. }
1442 {
1443   You~are~trying~to~use~a~template~directly~with~\UseInstance
1444   (or~a~similar~function),~but~the~template~types~do~not~match.
1445 }

1446 \msg_new:nnnn { template } { unknown-ad-hoc-keys }
1447 { Unknown~template~keys~"#1". }
1448 {
1449   LaTeX~was~asked~to~set~'ad-hoc'~template~keys,~but~the~keys~
1450   "#1"~are~not~known~here.
1451 }
1452 \msg_new:nnnn { template } { unknown-attribute }
1453 { The~template~attribute~'#1'~is~unknown. }
1454 {

```

```

1455     There~is~a~definition~in~the~current~template~reading\\
1456     \ \ \token_to_str:N \KeyValue {~#1~} \\
1457     but~there~is~no~key~called~'~#1~'.
1458 }
1459 \msg_new:nnnn { template } { unknown-choice }
1460 { The~choice~'~#2~'~was~not~declared~for~key~'~#1~'. }
1461 {
1462     The~key~'~#1~'~takes~a~fixed~list~of~choices~
1463     and~this~list~does~not~include~'~#2~'.
1464 }
1465 \msg_new:nnnn { template } { unknown-default-choice }
1466 { The~default~choice~'~#2~'~was~not~declared~for~key~'~#1~'. }
1467 {
1468     The~key~'~#1~'~takes~a~fixed~list~of~choices~
1469     and~this~list~does~not~include~'~#2~'.
1470 }
1471 \msg_new:nnnn { template } { unknown-equiv-keytype }
1472 { The~equivalent~key~type~'~#1~'~is~unknown. }
1473 {
1474     Valid~equivalent~key~types~are:\\
1475     --alias;\\
1476     --meta.\\
1477 }
1478 \msg_new:nnnn { template } { unknown-expansion }
1479 { The~expansion~type~'~#1~'~is~unknown. }
1480 {
1481     Key~values~can~only~be~expanded~using~one~of~the~pre~defined~methods:~
1482     n,~o,~V,~v,~e,~N~or~c.
1483 }
1484 \msg_new:nnnn { template } { unknown-instance }
1485 { The~instance~'~#2~'~of~type~'~#1~'~is~unknown. }
1486 {
1487     You~have~asked~to~use~an~instance~'~#2~',~
1488     but~this~has~not~been~created.
1489 }
1490 \msg_new:nnnn { template } { unknown-key }
1491 { Unknown~template~key~'~#1~'. }
1492 {
1493     The~key~'~#1~'~was~not~declared~in~the~interface~
1494     for~the~current~template.
1495 }
1496 \msg_new:nnnn { template } { unknown-keytype }
1497 { The~key~type~'~#1~'~is~unknown. }
1498 {
1499     Valid~key~types~are:\\
1500     --boolean;\\
1501     --choice;\\
1502     --commalist;\\
1503     --function;\\
1504     --instance;\\
1505     --integer;\\
1506     --length;\\
1507     --muskip;\\

```

```

1508     --real;\
1509     --skip;\
1510     --tokenlist.
1511 }
1512 \msg_new:nnnn { template } { unknown-type }
1513 { The~template~type~'#1'~is~unknown. }
1514 {
1515     A~template~type~needs~to~be~defined~with~\NewTemplateType
1516     prior~to~using~it.
1517 }
1518 \msg_new:nnnn { template } { unknown-template }
1519 { The~template~'#2'~of~type~'#1'~is~unknown. }
1520 {
1521     No~interface~has~been~declared~for~a~template~
1522     '#2'~of~template~type~'#1'.
1523 }

```

Information messages only have text: more text should not be needed.

```

1524 \msg_new:nnn { template } { declare-instance }
1525 { Declaring~instance~~'#1'~of~type~'#2'~\msg_line_context:. }
1526 \msg_new:nnn { template } { declare-template-code }
1527 { Declaring~code~for~template~'#2'~of~template~type~'#1'~\msg_line_context:. }
1528 \msg_new:nnn { template } { declare-template-interface }
1529 {
1530     Declaring~interface~for~template~'#2'~of~template~type~'#1'~
1531     \msg_line_context:.
1532 }
1533 \msg_new:nnn { template } { declare-type }
1534 { Declaring~template~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1535 \msg_new:nnn { template } { show-attribute }
1536 {
1537     The~template~'#2'~of~type~'#1'~has~
1538     \tl_if_empty:nTF {#4} { no~#3. } { #3 : #4 }
1539 }
1540 \msg_new:nnn { template } { show-values }
1541 {
1542     The~instance~'#2'~of~type~'#1'~has~
1543     \tl_if_empty:nTF {#3} { no~values. } { values: #3 }
1544 }

```

Also add template to the LaTeX messages.

```

1545 \prop_gput:Nnn \g_msg_module_type_prop { template } { LaTeX }

```

13.19 User functions

All simple translations.

```

\NewTemplateType
\DeclareTemplateInterface
\DeclareTemplateCode
\DeclareTemplateCopy
\EditTemplateDefaults
\UseTemplate
\DeclareInstance
\DeclareInstanceCopy
\EditInstance
\UseInstance

1546 \cs_new_protected:Npn \NewTemplateType #1#2
1547 { \__template_define_type:nn {#1} {#2} }
1548 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
1549 { \__template_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
1550 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
1551 { \__template_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
1552 \cs_new_protected:Npn \DeclareTemplateCopy #1#2#3
1553 { \__template_template_set_eq:nnn {#1} {#2} {#3} }

```

```

1554 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
1555 { \__template_edit_defaults:nnn {#1} {#2} {#3} }
1556 \cs_new_protected:Npn \UseTemplate #1#2#3
1557 { \__template_use_template:nnn {#1} {#2} {#3} }
1558 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
1559 { \__template_declare_instance:nnnn {#1} {#3} {#2} {#4} }
1560 \cs_new_protected:Npn \DeclareInstanceCopy #1#2#3
1561 { \__template_instance_set_eq:nnn {#1} {#2} {#3} }
1562 \cs_new_protected:Npn \EditInstance #1#2#3
1563 { \__template_edit_instance:nnn {#1} {#2} {#3} }
1564 \cs_new_protected:Npn \UseInstance #1#2
1565 { \__template_use_instance:nn {#1} {#2} }

```

(End of definition for `\NewTemplateType` and others. These functions are documented on page 3.)

`\DeclareTemplateInterfaceEquivalents`

The optional argument is filtered out directly, with everything else passed on.

```

1566 \NewDocumentCommand \DeclareTemplateInterfaceEquivalents { +m +m o +m }
1567 {
1568   \IfNoValueTF {#3}
1569   { \__template_declare_template_equiv:nnnn {#1} {#2} { } {#4} }
1570   {
1571     \__template_date_compare:nNnTF
1572     {
1573       \int_use:N \c_sys_year_int -
1574       \int_use:N \c_sys_month_int -
1575       \int_use:N \c_sys_day_int
1576     }
1577     >
1578     {#3}
1579     {
1580       \__template_declare_template_equiv:nnnn {#1} {#2}
1581       {
1582         \msg_error:nnnnn { template } { deprecated-equiv }
1583         {#3} { was }
1584       }
1585       {#4}
1586     }
1587     {
1588       \__template_declare_template_equiv:nnnn {#1} {#2}
1589       {
1590         \msg_warning:nnnnn { template } { deprecated-equiv }
1591         {#3} { will-be }
1592       }
1593       {#4}
1594     }
1595   }
1596 }

```

(End of definition for `\DeclareTemplateInterfaceEquivalents`. This function is documented on page 11.)

`\ShowTemplateCode` `\ShowTemplateDefaults` `\ShowTemplateInterface` `\ShowTemplateVariables` `\ShowInstanceValues`

The show functions are again just translation.

```

1597 \cs_new_protected:Npn \ShowTemplateCode #1#2
1598 { \__template_show_code:nn {#1} {#2} }
1599 \cs_new_protected:Npn \ShowTemplateDefaults #1#2

```

```

1600 { \__template_show_defaults:nn {#1} {#2} }
1601 \cs_new_protected:Npn \ShowTemplateInterface #1#2
1602 { \__template_show_keytypes:nn {#1} {#2} }
1603 \cs_new_protected:Npn \ShowTemplateVariables #1#2
1604 { \__template_show_vars:nn {#1} {#2} }
1605 \cs_new_protected:Npn \ShowInstanceValues #1#2
1606 { \__template_show_values:nn {#1} {#2} }

```

(End of definition for \ShowTemplateCode and others. These functions are documented on page 12.)

\IfInstanceExistsT More direct translation.

```

\IfInstanceExistsF 1607 \cs_new:Npn \IfInstanceExistsTF #1#2
\IfInstanceExistsTF { \__template_if_instance_exist:nnTF {#1} {#2} }
1608 { \__template_if_instance_exist:nnT {#1} {#2} }
1609 \cs_new:Npn \IfInstanceExistsT #1#2
1610 { \__template_if_instance_exist:nnT {#1} {#2} }
1611 \cs_new:Npn \IfInstanceExistsF #1#2
1612 { \__template_if_instance_exist:nnF {#1} {#2} }

```

(End of definition for \IfInstanceExistsT, \IfInstanceExistsF, and \IfInstanceExistsTF. These functions are documented on page 9.)

\KeyValue Simply dump the argument when executed: this should not happen.

```

1613 \cs_new_protected:Npn \KeyValue #1 {#1}

```

(End of definition for \KeyValue. This function is documented on page 4.)

\InstanceValue

```

1614 \cs_new:Npn \InstanceValue #1#2#3
1615 { \__template_instance_value:nnn {#1} {#2} {#3} }

```

(End of definition for \InstanceValue. This function is documented on page 9.)

\AssignTemplateKeys A short call to use a token register by proxy.

```

1616 \cs_new_protected:Npn \AssignTemplateKeys { \__template_assignments_pop: }

```

(End of definition for \AssignTemplateKeys. This function is documented on page 6.)

\SetKnownTemplateKeys A friendly wrapper, with some speed up for the common case of the third argument being empty.

\SetTemplateKeys

```

1617 \cs_new_protected:Npn \SetKnownTemplateKeys #1#2#3
1618 { \__template_keys_set_known:nnn {#1} {#2} {#3} }

1619 \cs_new_protected:Npn \SetTemplateKeys #1#2#3
1620 { \__template_keys_set:nnn {#1} {#2} {#3} }

```

(End of definition for \SetKnownTemplateKeys and \SetTemplateKeys. These functions are documented on page 6.)

\SetCurrentKnownTemplateKeys

```

1621 \cs_new_protected:Npn \SetCurrentKnownTemplateKeys #1
1622 {
1623   \exp_args:NVV \__template_keys_set_known:nnn
1624   \l__template_current_type_tl
1625   \l__template_current_template_tl
1626   {#1}
1627 }

```


\cs_if_exist:NTF	946, 969, 987, 130, 156, 169, 551, 613, 642, 654, 795, 820
\cs_if_exist_use:NTF	1038, 483
\cs_new:Npn	11, 1058, 21, 1233, 40, 46, 1607, 1609, 1611, 1614, 100, 441, 442, 655, 664
\cs_new_eq:NN	887, 898
\cs_new_protected:Npe	910, 366
\cs_new_protected:Npn	849, 857, 863, 873, 881, 889, 903, 922, 927, 932, 935, 938, 951, 960, 981, 994, 1006, 1012, 1018, 1023, 1045, 1053, 1063, 1078, 1084, 1104, 1121, 1127, 1133, 1144, 1150, 1162, 1172, 1191, 1193, 1204, 1209, 1215, 1221, 1227, 1234, 1236, 1245, 1247, 1261, 1263, 1272, 1281, 1290, 1298, 1546, 1548, 1550, 1552, 1554, 1556, 1558, 1560, 1562, 1564, 1597, 1599, 1601, 1603, 1605, 1613, 1616, 1617, 1619, 1621, 1629, 1630, 105, 111, 116, 117, 118, 128, 134, 141, 147, 154, 179, 187, 203, 211, 222, 232, 248, 258, 275, 281, 296, 314, 332, 344, 361, 387, 411, 443, 460, 462, 481, 492, 494, 496, 503, 511, 523, 549, 557, 576, 578, 592, 596, 611, 624, 638, 650, 652, 659, 678, 695, 716, 733, 740, 746, 751, 760, 762, 764, 773, 787, 810, 817, 831, 836
\cs_set:Npn	876, 419
\cs_set_eq:NN	989
\cs_set_protected:Npe	971
\cs_set_protected:Npn	1131, 1148, 107, 108, 113, 114, 415
\cs_show:N	1262, 45
D	
debug commands:	
\debug_resume:	185, 201, 209, 220
\debug_suspend:	181, 189, 205, 213
\DebugTemplatesOff	1629, 12
\DebugTemplatesOn	1629, 12
\DeclareInstance	1546, 9
\DeclareInstanceCopy	1546, 9
\DeclareTemplateCode	1350, 1406, 1432, 1546, 5
\DeclareTemplateCopy	1546, 7
\DeclareTemplateInterface	1546, 4
\DeclareTemplateInterfaceEquivalents	1566, 11
dim commands:	
\dim_new:N	93
\l_tmpa_dim	5
E	
\EditInstance	1546, 10
\EditTemplateDefaults	1546, 10
\EndModuleRelease	1634
exp commands:	
\exp_after:wN	1050, 1192, 1199, 376, 833
\exp_args:Ne	1164
\exp_args:NNe	56
\exp_args:NnnV	1256
\exp_args:No	1246
\exp_args:NV	878
\exp_args:NVV	1623
\exp_not:N	913, 915, 916, 918, 919, 973, 974, 975, 977, 10, 15, 1091, 1099, 1139, 1154, 1179, 1186, 368, 369, 370, 371, 381, 387, 390, 392, 393, 397, 399, 402, 407, 633, 793, 810, 811
\exp_not:n	976, 1092, 1093, 1100, 1138, 1140, 1141, 1154, 1157, 1158, 1179, 1180, 1186, 1187, 373, 520
\ExplSyntaxOff	1635
\ExplSyntaxOn	6
F	
fp commands:	
\l_tmpa_fp	5
I	
\IfInstanceExistsF	1607, 9
\IfInstanceExistsT	1607, 9
\IfInstanceExistsTF	1607, 9
\IfNoValueTF	1568
\IncludeInRelease	1631
\InstanceValue	1614, 9
int commands:	
\int_compare:nNnTF	23, 25, 27, 31, 36, 121
\int_compare:nTF	284
\int_new:N	94
\int_set:Nn	283
\int_use:N	1573, 1574, 1575
\l_tmpa_int	5
iow commands:	
\iow_term:n	109
\item	7
K	
kernel internal commands:	
_kernel_quark_new_conditional:Nn	104
keys commands:	
\keys_set_known:nnN	1254
keyval commands:	
\keyval_map_break:n	508

\keyval_map_inline:nnn	498
\keyval_parse:NNn	306
\keyval_parse:nnn	868, 451, 562, 682, 768
\KeyValue	1193, 1456, <u>1613</u> , 162, <u>42</u>
M	
\message	3
msg commands:	
\msg_error:nn	358
\msg_error:nnn	869, 904, 933, 1041, 1202, 1241, 138, 145, 150, 278, 338, 382, 404, 461, 473, 486, 507, 577, 590, 608, 635, 647, 692, 782, 804
\msg_error:nnnn	992, 1004, 1115, 1219, 1225, 1311, 132, 158, 292, 710, 738, 748
\msg_error:nnnnn	1582, 124, 571
\msg_info:nnnn	947, 970, 988, 192, 286, 552
\msg_line_context:	1072, 1230, 1362, 1368, 1376, 1383, 1390, 1397, 1404, 1411, 1419, 1525, 1527, 1531, 1534
\g_msg_module_type_prop	1545
\msg_new:nnn	1524, 1526, 1528, 1533, 1535, 1540
\msg_new:nnnn	1313, 1321, 1326, 1332, 1341, 1347, 1354, 1361, 1367, 1373, 1382, 1389, 1396, 1403, 1410, 1418, 1425, 1434, 1440, 1446, 1452, 1459, 1465, 1471, 1478, 1484, 1490, 1496, 1512, 1518
\msg_show:nnnnn	1303
\msg_show:nnnnnn	1292
\msg_show_item_unbraced:nn	1296, 1308
\msg_warning:nnnnn	1590
muskip commands:	
\muskip_new:N	95
\l_tmpa_muskip	5
N	
\NewDocumentCommand	1566
\NewModuleRelease	7
\NewTemplateType	1437, 1515, <u>1546</u> , 3
\NoValue	44
P	
prg commands:	
\prg_generate_conditional_variant:Nnn	166
\prg_new_conditional:Npnn	9, 160, 167, 173
\prg_return_false:	28, 32, 37, 164, 171, 177
\prg_return_true:	28, 32, 37, 163, 170, 176
prop commands:	
\prop_clear:N	230, 240, 256, 266, 273, 302, 303, 304
\prop_clear_new:N	206
\prop_gclear:N	193
\prop_gclear_new:N	182, 214, 217
\prop_get:NnN	852, 859, 866, 999, 1020, 1112, 120, 705, 708, 735, 775
\prop_get:NnNTF	842, 892, 1027, 1123, 1196, 467, 582, 686, 797, 823
\prop_gput:Nnn	1545, 288
\prop_gset_eq:NN	183, 196, 215, 218
\prop_if_exist:NTF	190, 224, 234, 250, 260, 267
\prop_if_in:NnTF	143, 277, 699, 703
\prop_item:Nn	1061
\prop_map_function:NN	1296, 1307
\prop_map_inline:Nn	565
\prop_new:N	75, 86, 88, 90, 91, 195
\prop_pop:NnNTF	1025
\prop_put:Nnn	884, 918, 925, 930, 966, 352, 364, 513, 516, 519, 644, 656, 661, 681, 744
\prop_remove:Nn	588
\prop_set_eq:NN	207, 227, 237, 253, 263, 270
Q	
quark commands:	
\q_nil	162, 175, 834, 836
\quark_new:N	103
\q_stop	16, 18, 40, 43, 46, 162, 175, 793, 811, 834, 836
quark internal commands:	
\q_template_nil	1200, 103, 16
R	
regex commands:	
\regex_match:nnTF	534
S	
scan commands:	
\scan_new:N	101, 102
scan internal commands:	
\s__template_mark	1200, 101, 16
\s__template_stop	1050, 1053, 1200, 1204, 102, 377, 388, 399, 420, 434, 442, 594, 597, 620, 625, 16
\section	2
seq commands:	
\seq_clear:N	246, 305, 767
\seq_const_from_clist:Nn	56, 73

\seq_gclear_new:N	198
\seq_gset_eq:NN	199
\seq_if_exist:NnTF	241
\seq_if_in:NnTF	136, 149, 470, 505
\seq_map_break:	340, 431
\seq_map_function:NN	1015, 321, 437
\seq_new:N	87, 89
\seq_put_right:Nn	883, 891, 354, 515
\seq_remove_all:Nn	851, 865
\seq_set_eq:NN	243
\SetCurrentKnownTemplateKeys	1621, 6
\SetKnownTemplateKeys	1617, 6
\SetTemplateKeys	1617, 6
\ShowInstanceValues	1597, 12
\ShowTemplateCode	1597, 12
\ShowTemplateDefaults	1597, 12
\ShowTemplateInterface	1597, 12
\ShowTemplateVariables	1597, 12
skip commands:	
\skip_new:N	96
\l_tmpa_skip	5
str commands:	
\str_case:nn	666
\str_if_eq:nnTF	1217, 162, 175, 334, 355, 720, 776
\str_if_eq_p:nn	1033, 1034, 1035, 568, 569
\str_if_in:nnTF	532
\str_new:N	85
\str_set:Nn	815
sys commands:	
\c_sys_day_int	1575
\c_sys_month_int	1574
\c_sys_year_int	1573
T	
template internal commands:	
__template_assign_boolean:	1078, 1078
__template_assign_boolean_aux:n	1078, 1081, 1082, 1084
__template_assign_choice:	1104, 1104
__template_assign_choice_-	aux:nTF 1104, 1106, 1109, 1121, 1126
__template_assign_function:	1127, 1127
__template_assign_function_-	aux:N 1127, 1130, 1131, 1133
__template_assign_instance:	1144, 1144
__template_assign_instance_-	aux:N 1144, 1147, 1148, 1150
__template_assign_variable:	1039, 1162, 1162
__template_assign_variable:n	1162, 1164, 1172
__template_assignments_pop:	1233, 1233, 1616
__template_assignments_push:n	975, 1234, 1234
\l__template_assignments_tl	976, 1014, 1089, 1097, 1124, 1135, 1152, 1177, 1184, 1233, 1235, 1258, 76, 15
\c__template_code_root_tl	946, 948, 949, 977, 1075, 1262, 48, 130, 551, 554, 14
__template_convert_to_assignments:N	968, 1012, 1012, 1070, 1257
__template_convert_to_assignments_-	aux:n 1012, 1016, 1018
__template_convert_to_assignments_-	aux:nn 1012, 1021, 1023, 1044
\l__template_current_template_tl	974, 1074, 1625, 99, 16
\l__template_current_type_tl	973, 1073, 1624, 98, 16
__template_date_compare:nNn	9
__template_date_compare:nNnTF	9, 1571
__template_date_compare_aux:nNn	9, 10, 11
__template_date_compare_-	auxi:nnnNnnn 9, 15, 21
__template_date_compare_auxii:w	9, 16, 18, 40
__template_date_compare_-	auxiii:w 9, 43, 46
__template_debug:n	107, 113, 116, 116
__template_debug_off:	1630, 105, 111
__template_debug_on:	1629, 105, 105
__template_debug_typeout:n	1071, 1229, 108, 114, 116, 117
__template_declare_instance:nnnn	951, 951, 1559
__template_declare_instance_-	aux:nnnn 951, 957, 960, 1009
__template_declare_template_-	code:nnnn 523, 535, 537, 542, 549
__template_declare_template_-	code:nnnnn 1551, 523, 523
__template_declare_template_-	equiv:nnnn 1569, 1580, 1588, 443, 443
__template_declare_template_-	keys:nnnn 1549, 296, 296
__template_declare_type:nn	275, 279, 281
\l__template_default_tl	77, 15

```

\c__template_defaults_root_tl ...
    ..... 49, 182, 183, 225, 228, 14
\__template_define_type:nn .....
    ..... 1547, 275, 275
\__template_edit_defaults:nnn ...
    ..... 1555, 751, 751
\__template_edit_instance:nnn ...
    ..... 994, 994, 1563
\__template_edit_instance_-
    aux:nnnn ..... 1001, 1006, 1011
\__template_edit_instance_-
    aux:nnnnn ..... 994
\l__template_error_bool .....
    ..... 962, 964, 78,
    317, 323, 339, 368, 381, 465, 781, 15
\__template_execute_if_arg_-
    agree:nnTF .... 118, 118, 300, 527
\__template_execute_if_code_-
    exist:nnTF 953, 1065, 1283, 128, 128
\__template_execute_if_keyname_-
    free:nTF ..... 147, 147, 153, 325
\__template_execute_if_keys_-
    exist:nnTF ..... 154
\__template_execute_if_keytype_-
    exist:nTF .... 134, 134, 140, 319
\__template_execute_if_type_-
    exist:nTF .. 141, 141, 298, 445, 525
\__template_find_global: .....
    ..... 1037, 1045, 1045
\__template_find_global_aux:w ...
    ..... 1045, 1050, 1053
\l__template_global_bool .. 1047,
    1056, 1080, 1129, 1146, 1168, 79, 15
\__template_if_instance_exist:nn 167
\__template_if_instance_exist:nnTF
    ..... 983, 996,
    1060, 1223, 1300, 1608, 1610, 1612, 167
\__template_if_key_value:n . 160, 166
\__template_if_key_value:nTF ...
    ..... 1086, 1174, 160
\__template_if_keys_exist:nnTF ..
    ..... 1265, 1274, 154, 447, 529, 753
\__template_if_use_template:n .. 173
\__template_if_use_template:nTF .
    ..... 1211, 173
\__template_implement_choice_-
    elt:n ..... 683, 716, 746
\__template_implement_choice_-
    elt:nnn ..... 684, 716, 716
\__template_implement_choice_-
    elt_aux:n ..... 716, 722, 730, 733
\__template_implement_choice_-
    elt_aux:nnn ..... 716, 721, 728, 740
\__template_implement_choices:nn
    ..... 651, 678, 678
\__template_implement_choices_-
    default: ..... 678, 688, 695
\__template_instance_set_eq:nnn .
    ..... 981, 981, 1561
\__template_instance_value:nnn ..
    ..... 1058, 1058, 1615
\c__template_instances_root_tl ..
    ..... 969,
    971, 987, 989, 990, 1231, 50, 169, 14
\l__template_key_name_tl .....
    ..... 839, 841, 842, 845, 851, 852,
    854, 859, 865, 866, 875, 883, 884,
    891, 892, 919, 925, 930, 1029, 1107,
    1110, 1112, 1116, 80, 326, 352,
    354, 364, 375, 390, 397, 402, 468,
    474, 513, 515, 516, 519, 520, 580,
    583, 588, 645, 657, 662, 681, 686,
    698, 702, 705, 708, 711, 712, 735,
    738, 743, 749, 797, 814, 823, 834, 33
\c__template_key_order_root_tl ..
    ..... 52, 198, 199, 241, 244, 14
\l__template_key_order_seq .....
    ..... 968, 1070, 87, 149,
    200, 243, 246, 305, 354, 505, 515, 15
\__template_key_to_value: .....
    ..... 1088, 1176, 1191, 1191
\__template_key_to_value_auxi:w .
    ..... 1191, 1192, 1193
\__template_key_to_value_auxii:w
    ..... 1191, 1199, 1204
\__template_keys_set:nnn .....
    ..... 1236, 1236, 1620
\__template_keys_set_known:nnn ..
    ..... 1236, 1238, 1245, 1618, 1623
\__template_keys_set_known_-
    aux:nnn ..... 1236, 1246, 1247
\l__template_keytype_arg_tl ....
    ..... 1140, 1157, 82,
    336, 349, 350, 357, 414, 428, 680, 15
\l__template_keytype_tl .....
    ..... 860, 895, 908, 911, 1033,
    1034, 1035, 1038, 81, 319, 334,
    348, 355, 413, 424, 484, 487, 514,
    584, 586, 613, 615, 666, 821, 826, 15
\c__template_keytypes_arg_seq ...
    ..... 73, 321, 437, 14
\l__template_keytypes_prop .....
    .. 842, 859, 1020, 1112, 1277, 86,
    197, 237, 240, 303, 352, 467, 513,
    565, 582, 588, 705, 735, 775, 797, 15
\c__template_keytypes_root_tl 51,
    156, 190, 193, 195, 196, 235, 238, 14

```

\c__template_keytypes_seq	56, 136, 470, 14	__template_parse_values_-	
__template_map_var_type:		alias:nn	849, 787
..... 915, 1166, 643, 664, 664		__template_parse_values_alias_-	
\l__template_msg_prop		aux:nn	855, 857, 878, 787
..... 91, 216, 263, 266, 519, 823, 15		__template_parse_values_elt:NNn	
\c__template_msg_root_tl		 769, 773, 773	
..... 53, 214, 215, 261, 264, 14		__template_parse_values_elt:Nnn	
__template_parse_equiv_alias:nnn	492	 770, 777, 787, 787	
__template_parse_equiv_alias:nnnn	462	__template_parse_values_elt_-	
__template_parse_equiv_auxi:nnnn	462, 493, 495, 496	auxi:n	787
__template_parse_equiv_auxii:n	462, 499, 500, 503	__template_parse_values_elt_-	
__template_parse_equiv_auxiii:nnnn	462, 501, 511	auxi:w	787, 791, 810
__template_parse_equiv_elt:n	452, 460, 460	__template_parse_values_elt_-	
__template_parse_equiv_elt:nnnn	453, 462, 462	auxii:nn	844, 861, 799, 817
__template_parse_equiv_elt_-		__template_parse_values_elt_-	
aux:nnn	462, 476, 478, 481	auxiii:Nn	787, 800, 831
__template_parse_equiv_meta:nnn	494	__template_parse_values_elt_-	
__template_parse_equiv_meta:nnnn	462	auxiv:w	787, 833, 836
__template_parse_keys_elt:n	307, 314, 314, 363, 23	__template_parse_values_meta:nn	863, 787
__template_parse_keys_elt:nn	307, 361, 361	__template_parse_values_meta_-	
__template_parse_keys_elt_aux:	314, 327, 344	elt:nnnn	870, 873, 787
__template_parse_keys_elt_aux:n	314, 322, 332	__template_parse_values_set:N	887, 888, 787
__template_parse_known_values:nnn	1256, 760, 762	__template_parse_values_set:n	881, 886, 887, 787
__template_parse_unknown_-		__template_parse_vars_choice:nnn	650, 650
error:nn	932, 934, 761	__template_parse_vars_elt:n	563, 576, 576
__template_parse_unknown_-		__template_parse_vars_elt:nnn	563, 578, 578
store:nn	935, 937, 763	__template_parse_vars_elt_-	
__template_parse_values:nnn	963, 1069, 756, 760, 760	auxi:nn	578, 587, 592
__template_parse_values:NNnnn	760, 761, 763, 764	__template_parse_vars_elt_-	
__template_parse_values_add:N	898, 899, 787	auxii:nw	578, 594, 596
__template_parse_values_add:n	889, 897, 898, 787	__template_parse_vars_elt_-	
__template_parse_values_add_-		auxiii:nnn	578, 600, 604, 611, 623
commalist:n	922	__template_parse_vars_elt_-	
__template_parse_values_add_-		auxiv:nnw	578, 619, 624
tokenlist:n	927	__template_parse_vars_elt_-	
		auxv:nnn	578, 628, 632, 638, 649
		__template_parse_vars_function:nnn	650, 652
		__template_parse_vars_instance:nnn	650, 659
		__template_quark_if_nil:N	104
		__template_quark_if_nil:NTF	1206
		__template_quark_if_nil:nTF	104
		__template_quark_if_nil_p:n	104
		__template_recover_defaults:nn	940, 955, 1067, 1267, 222, 222, 559, 755
		__template_recover_keytypes:nn	942, 1276, 222, 232, 449, 560, 766

```

\__template_recover_values:nn ...
    ..... 985, 998, 1302, 222, 248
\__template_recover_vars:nn ...
    ..... 944, 956, 1008,
    1068, 1255, 1285, 222, 258, 450, 561
\c__template_restrict_root_tl ... 14
\__template_show:Nnnn ...
    ..... 1263, 1268, 1277, 1286, 1290
\__template_show_code:nn ...
    ..... 1261, 1261, 1598
\__template_show_defaults:nn ...
    ..... 1263, 1263, 1600
\__template_show_keytypes:nn ...
    ..... 1263, 1272, 1602
\__template_show_values:nn ...
    ..... 1298, 1298, 1606
\__template_show_vars:nn ...
    ..... 1263, 1281, 1604
\__template_split_keytype:n ...
    ..... 316, 366, 366, 464
\__template_split_keytype_arg:n .
    ..... 1030, 1114,
    407, 411, 411, 440, 586, 707, 737, 819
\__template_split_keytype_arg-
aux:n ..... 411, 415, 438, 441
\__template_split_keytype_arg-
aux:w ..... 411, 419, 434, 442
\__template_split_keytype_aux:w .
    ..... 366, 376, 387, 399
\__template_store_defaults:nn ...
    ..... 941, 179, 179, 308, 757
\__template_store_key_implementation:nnn
    ..... 531, 557, 557
\__template_store_keytypes:nn ...
    ..... 943, 179, 187, 309, 455
\__template_store_values:nn ...
    ..... 967, 986, 179, 203
\__template_store_vars:nn ...
    ..... 945, 179, 211, 310, 456, 564
\__template_template_set_eq:nnn .
    ..... 938, 938, 1553
\__template_tmp:n .. 876, 877, 100, 100
\l__template_tmp_clist .....
    .. 92, 680, 689, 691, 718, 725, 727, 16
\l__template_tmp_dim ..... 93, 16
\l__template_tmp_int .....
    ..... 94, 283, 284, 287, 289, 293, 16
\l__template_tmp_muskip ..... 95, 16
\l__template_tmp_skip ..... 96, 16
\l__template_tmp_tl .....
    . 843, 853, 854, 867, 871, 877, 879,
    893, 894, 913, 916, 919, 924, 925,
    929, 930, 1000, 1002, 1020, 1021,
    1113, 1114, 1123, 1124, 1195, 1196,
    1202, 1254, 1256, 97, 120, 121, 125,
    346, 353, 369, 370, 371, 377, 468,
    471, 687, 697, 698, 699, 701, 702,
    703, 706, 707, 709, 736, 737, 742,
    744, 775, 776, 798, 819, 824, 825, 16
\g__template_type_prop .....
    ..... 75, 120, 143, 277, 288, 14
\__template_use_instance:nn ...
    ..... 1156, 1209, 1209, 1565
\__template_use_instance-
aux:nNnnn ..... 1209, 1212, 1215
\__template_use_instance_auxi:nn
    ..... 1209, 1213, 1221
\__template_use_instance-
auxii:nn ..... 1209, 1224, 1227
\__template_use_template:nnn ...
    ..... 1063, 1063, 1218, 1557
\l__template_value_exp_str .....
    ..... 85, 795, 805, 815, 828, 15
\l__template_value_order_seq ...
    . 851, 865, 883, 891, 1257, 89, 767, 15
\l__template_value_tl .....
    ..... 1025, 1086, 1093, 1099,
    1107, 1117, 1141, 1158, 1174, 1180,
    1187, 1192, 1197, 1199, 1207, 83, 15
\l__template_values_prop .....
    . 884, 892, 918, 925, 930, 966, 999,
    1025, 1268, 1307, 88, 184, 208, 227,
    230, 253, 256, 302, 364, 686, 708, 15
\c__template_values_root_tl .....
    ..... 1061, 54, 206, 207, 251, 254, 14
\l__template_var_tl .....
    ... 1027, 1048, 1050, 1055, 1092,
    1100, 1138, 1154, 1179, 1186, 84, 15
\l__template_vars_prop .....
    ..... 852, 866, 1027, 1123, 1196,
    1286, 90, 219, 270, 273, 304, 516,
    644, 656, 661, 681, 699, 703, 744, 15
\c__template_vars_root_tl .....
    ..... 55, 217, 218, 268, 271, 14
tl commands:
\c_space_tl . 1107, 1110, 698, 702, 743
\tl_clear:N . 894, 1014, 1249, 375, 414
\tl_const:Nn 48, 49, 50, 51, 52, 53, 54, 55
\tl_head:w ..... 162, 175
\tl_if_blank:nTF .....
    ..... 42, 422, 426, 599, 602, 627, 630
\tl_if_blank_p:n ..... 1251
\tl_if_empty:nTF .. 1239, 336, 349, 402
\tl_if_empty:nTF . 838, 1538, 1543, 517
\tl_if_in:nnTF ... 1048, 371, 395, 417
\tl_if_novalue_p:n ..... 1252
\tl_if_single:nTF ..... 640

```

